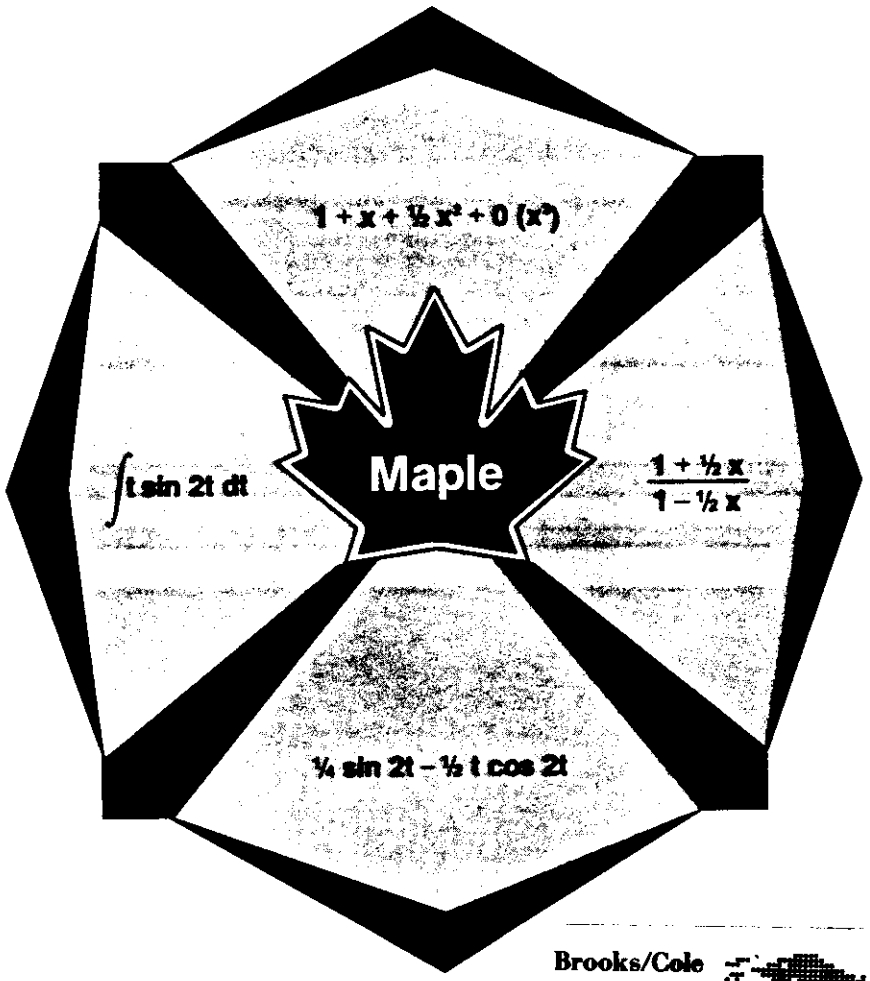
MAPLE

Bruce W. Char
Keith O. Geddes
Gaston H. Gonnet
Michael B. Monagan
Stephen M. Watt

Reference Manual

5th Edition



Brooks/Cole
Publishing
Company

M A P L E

Reference Manual

Fifth Edition

Bruce W. Char

Keith O. Geddes

Gaston H. Gonnet

Michael B. Monagan

Stephen M. Watt

March 1988

Symbolic Computation Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Published by
WATCOM Publications Limited
415 Phillip Street
Waterloo, Ontario, CANADA
N2L 3X2

(519) 886-3700
Telex 06-955458
FAX (519) 747-4971

Canada Cataloguing in Publication Data

Main entry under title:
Maple reference manual
5th ed.
Fourth ed. published as: Maple user's guide.

Includes index.
ISBN 0-921329-15-6

1. MAPLE (Computer system).

I. Char, Bruce W.
II. Title: Maple user's guide

QA76.8.M36M36 1988
512.9'2'028553 C88-093864-1

© by WATCOM Publications Limited, 1985,1988

All rights reserved. No part of this publication may be reproduced or used in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping or information storage and retrieval systems - without written permission of WATCOM Publications Limited.

Printed and bound in Canada
by John Deyell Company

Disclaimer

WATCOM (The WATCOM Group Inc. and all if its subsidiaries) makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall WATCOM, its employees, its contractors, or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands or claim for lost profits, fees or expenses of any nature or kind.

WATCOM Series in Computer Science and Computer Applications

WATFILE/Plus
Tutorial and Reference
by Wilkinson

Waterloo Pascal
Primer and Reference
by Boswell, Grove and Mackie

WATCOM APL
Reference
by Wilson and Wilkinson

WATCOM BASIC
A Structured Programming Approach
Primer and Reference
by Graham, McPhee and Welch

WATCOM COBOL
Tutorial and Reference
by Dirksen and Welch

WATCOM COBOL
A Structured Programming Approach
Primer
by Cowan, Dirksen and Graham

WATFOR-77
Language Reference
by Coschi and Schueler

WATCOM Pascal
Primer and Reference
by Boswell, Mackie and Grove

WATCOM GKS
Graphics
Tutorial and Reference
by Yach

Waterloo MacJANET
User's Guide
by Durance and Mackie

WATCOM Express C
User's Guide
by Hillis

WATCOM C
Library Reference
by Crigger and Welch

Waterloo C
Run-time Library Reference
by Carmody and Mulholland

Waterloo C
Panel Library Reference
by Simmons

Preface

The design and implementation of the Maple system is an on-going project of the Symbolic Computation Group at the University of Waterloo. This *fifth edition* of the Reference Manual corresponds with the version 4.2 release of the Maple system. There will be updates to this edition, and new editions, for future versions of Maple. The on-line **help** subsystem can be invoked from within a Maple session to view documentation on specific topics. In particular, the command **help(updates);** points the user to documentation updates for each new version of Maple.

The Maple project was first conceived in the autumn of 1980 growing out of discussions on the state of symbolic computation at the University of Waterloo. The authors wish to acknowledge many fruitful discussions with colleagues at the University of Waterloo, particularly Morven Gentleman, Michael Malcolm, and Frank Tompa. It was recognized in these discussions that none of the locally-available systems for symbolic computation provided the environment nor the facilities that should be expected for symbolic computation in the 1980's. We concluded that since the basic design decisions for the then-current symbolic systems such as ALTRAN[Hal71], CAMAL[Bou71], REDUCE[Hea71], and MACSYMA[Mar71] were made more than a decade earlier, it would be wise to design a new system from scratch taking advantage of the software engineering technology that had become available since then, as well as drawing from the lessons of experience.

Maple's basic features (e.g., elementary data structures, input/output, arithmetic with numbers, and elementary simplification) are coded in a systems programming language for efficiency. For users, there is a high-level language with a modern syntax more suitable for describing algebraic algorithms. An important property of Maple is that most of the algebraic facilities in the system are implemented using the high-level user language. The basic system, or kernel, is sufficiently compact and efficient to be practical to use in a time-sharing environment. Library functions are loaded into the system as required, adding to the system facilities such as polynomial factorization, equation solving, indefinite integration, matrix manipulation functions, etc. The modularity of this design allows users to consume computer resources proportional to the algebraic facilities actually being used.

The system kernel is written in macros which can be translated by a locally-developed macro processor (called Margay) into versions of the kernel in (dialects of) the C programming language on various different operating systems. Operating systems currently supporting a Maple implementation include UNIX (and various UNIX-like systems), DEC VMS, IBM VM/CMS, Amdahl UTS, Amiga DOS, and Macintosh MPW.

Acknowledgements

The *Maple Reference Manual* has benefited from comments given to us by students, faculty, and staff at the University of Waterloo, and by users at distribution sites throughout the world. These "friends of Maple" are too numerous to list here. We hope they will feel that their assistance has been acknowledged when they find their suggestions incorporated into the present edition. We would like to recognize specifically Stan Devitt, Greg Fee, and Benton Leong.

The Maple project has received support from various sources, including the Academic Development Fund of the University of Waterloo, the Natural Sciences and Engineering Research Council of Canada, Digital Equipment Corporation, the Sloan Foundation, and the Information Technology Research Centre of Ontario.

Table of Contents

Preface III
Acknowledgements IV
1. INTRODUCTION 1
2. LANGUAGE ELEMENTS 3
2.1. Character Set 3
2.2. Tokens 4
2.3. Blanks, Lines, Comments, and Continuation 6
2.4. Files 8
3. STATEMENTS AND EXPRESSIONS 10
3.1. Types of Statements 10
3.1.1. Assignment Statement 10
3.1.2. Expression 10
3.1.3. Read Statement 10
3.1.4. Save Statement 10
3.1.5. Selection Statement 11
3.1.6. Repetition Statement 12
3.1.7. Empty Statement 13
3.1.8. Quit Statement 13
3.2. Expressions 13
3.2.1. Constants 14
3.2.2. Names and Strings 16
3.2.3. Expression Sequences 20
3.2.4. Sets and Lists 21
3.2.5. Algebraic Operators 23
3.2.6. Neutral Operators 25

3.2.7. Relations and Logical Operators 26
3.2.8. Ranges 27
3.2.9. Selection Operation 29
3.2.10. Unevaluated Expressions 30
3.2.11. Procedures and Functions 30
3.2.12. Functional Operators 32
3.2.13. Precedence of Programming-Language Operators 34
3.3. Sample Maple Session 35
4. DATA TYPES AND FORMAL SYNTAX 42
4.1. Data Types 42
4.1.1. Integer 42
4.1.2. Fraction 42
4.1.3. Floating-Point Number 43
4.1.4. String 44
4.1.5. Indexed Name 44
4.1.6. Addition, Multiplication, and Exponentiation 45
4.1.7. Series 45
4.1.8. Relation 46
4.1.9. Boolean Expression 47
4.1.10. Range 47
4.1.11. Expression Sequence 48
4.1.12. Set and List 49
4.1.13. Table and Array 49
4.1.14. Procedure Definition 50
4.1.15. Unevaluated Function Invocation 51
4.1.16. Unevaluated Concatenation 52
4.1.17. Unevaluated Expression 52
4.2. Formal Syntax 52

5. ARRAYS AND TABLES 56
5.1. Overview 56
5.2. Creating Tables 57
5.2.1. Explicit Table Creation 57
5.2.2. Implicit Table Creation 59
5.3. Evaluation Rules for Tables and Table Components 60
5.3.1. Evaluation to a Table 60
5.3.2. Evaluating Table Components 61
5.3.3. Assigning and Unassigning Table Components 63
5.4. Tables as Objects 64
5.4.1. Copying Tables 64
5.4.2. Tables Local to a Procedure 65
5.4.3. Tables as Parameters 66
5.4.4. Automatic Loading of Tables 66
5.5. Indexing Functions 67
5.5.1. The Purpose of Indexing Functions 67
5.5.2. Indexing Functions Known to the Basic System 68
5.5.3. User-Defined Indexing Functions 70
6. PROCEDURES 74
6.1. Procedure Definitions 74
6.2. Parameter Passing 76
6.3. Local Variables 78
6.4. Options 79
6.5. Remember Tables 80
6.6. Assigning Values to Parameters 83
6.7. Error Returns and Explicit Returns 85
6.8. Boolean Procedures 87
6.9. Reading and Saving Procedures 88

7. OPERATORS 91
7.1. Operator definition 91
7.2. Syntactic definition 92
7.3. Semantic definition 93
7.3.1. Application versus composition 94
7.4. Partial definition of operators 95
7.5. Example: The differentiation operator D 96
8. INTERNAL REPRESENTATION AND MANIPULATION 99
8.1. Internal Organization 99
8.2. Internal Representation of Data Types 101
8.3. The Use of Hashing in Maple 107
8.3.1. The Simplification Table 108
8.3.2. The Name Table 109
8.3.3. Put Tables 110
8.3.4. Remember Tables 110
8.3.5. Arrays and Tables in the Maple Language 111
8.4. Portability of the Maple System 111
9. THE MAPLE LIBRARY 113
9.1. Standard library functions 116
9.1.1. abs 116
9.1.2. anames 116
9.1.3. arc-trigonometric functions 116
9.1.4. arc-hyperbolic functions 116
9.1.5. array 116
9.1.6. assign 117
9.1.7. assigned 118
9.1.8. asympt 118
9.1.9. bernoulli 119
9.1.10. binomial 119
9.1.11. cat 120
9.1.12. chebyshev 120

9.1.13. coeff 121
9.1.14. coeffs 121
9.1.15. collect 122
9.1.16. content and primpart 122
9.1.17. Content and Primpart 123
9.1.18. convert 124
9.1.18.1. convert/`+` and convert/`*` 125
9.1.18.2. convert/array 125
9.1.18.3. convert/base 126
9.1.18.4. convert/binary 126
9.1.18.5. convert/confrac 127
9.1.18.6. convert/decimal 128
9.1.18.7. convert/degrees 128
9.1.18.8. convert/eqnlist 128
9.1.18.9. convert/equality; convert/lessthan; convert/lessequal 129
9.1.18.10. convert/exp 129
9.1.18.11. convert/expsincos 130
9.1.18.12. convert/factorial 130
9.1.18.13. convert/float 130
9.1.18.14. convert/GAMMA 131
9.1.18.15. convert/hex 131
9.1.18.16. convert/horner 131
9.1.18.17. convert/hostfile 132
9.1.18.18. convert/list 132
9.1.18.19. convert/listlist 133
9.1.18.20. convert/ln 133
9.1.18.21. convert/metric 134
9.1.18.22. convert/name 134
9.1.18.23. convert/octal 134
9.1.18.24. convert/parfrac 135

9.1.18.25. convert/polar 135
9.1.18.26. convert/polynom 136
9.1.18.27. convert/radians 136
9.1.18.28. convert/radical 136
9.1.18.29. convert/rational 137
9.1.18.30. convert/ratpoly 137
9.1.18.31. convert/RootOf 138
9.1.18.32. convert/series 139
9.1.18.33. convert/set 139
9.1.18.34. convert/sincos 140
9.1.18.35. convert/sqrfree 140
9.1.19. copy 141
9.1.20. define 141
9.1.20.1. define/group 142
9.1.20.2. define/linear 143
9.1.20.3. define/properties 143
9.1.21. degree and ldegree 144
9.1.22. diff 144
9.1.23. divide 145
9.1.24. Divide 145
9.1.25. dsolve 146
9.1.26. erf 147
9.1.27. ERROR 147
9.1.28. euler 148
9.1.29. eval 148
9.1.30. evala 149
9.1.31. evalb 150
9.1.32. evalc 150
9.1.33. evalf 151
9.1.34. evaln 152
9.1.35. exp 152

9.1.36. expand 153
9.1.37. factor 153
9.1.38. Factor 154
9.1.39. frontend 154
9.1.40. fsolve 155
9.1.41. GAMMA 156
9.1.42. gc 156
9.1.43. gcd and lcm 157
9.1.44. Gcd 157
9.1.45. gcdex 158
9.1.46. Gcdex 159
9.1.47. genpoly 159
9.1.48. has 160
9.1.49. hastype 160
9.1.50. help 161
9.1.51. hyperbolic functions 161
9.1.52. icontent 161
9.1.53. ifactor 162
9.1.54. igcd and ilcm 163
9.1.55. igcdex 163
9.1.56. indets 163
9.1.57. indices 164
9.1.58. int 164
9.1.59. interp 165
9.1.60. Interp 166
9.1.61. irem and iquo 166
9.1.62. isolve 167
9.1.63. isprime 167
9.1.64. isqrt 168
9.1.65. ithprime 168
9.1.66. lcoeff and tcoeff 168

9.1.67. length 169
9.1.68. lexorder 169
9.1.69. limit 170
9.1.69.1. limit/directional 170
9.1.69.2. limit/multidirectional 171
9.1.69.3. limit/values 171
9.1.70. ln 172
9.1.71. lprint 172
9.1.72. map 173
9.1.73. max and min 173
9.1.74. maxnorm 174
9.1.75. member 174
9.1.76. mod, modp, and mods 174
9.1.77. msolve 176
9.1.78. nextprime and prevprime 177
9.1.79. normal 177
9.1.80. numer and denom 178
9.1.81. op and nops 178
9.1.82. order 179
9.1.83. plot 179
9.1.83.1. plot/device 180
9.1.83.2. plot/function 181
9.1.83.3. plot/infinity 182
9.1.83.4. plot/line 182
9.1.83.5. plot/multiple 182
9.1.83.6. plot/option 183
9.1.83.7. plot/parametric 184
9.1.83.8. plot/point 184
9.1.83.9. plot/polar 184
9.1.83.10. plot/range 185
9.1.83.11. plot/replot 185

9.1.83.12. plot/setup 186
9.1.83.13. plot/spline 187
9.1.83.14. plot/structure 187
9.1.84. prem and sprem 188
9.1.85. Prem and Sprem 189
9.1.86. print 190
9.1.87. product 190
9.1.88. Psi 191
9.1.89. radsimp 191
9.1.90. rand 191
9.1.91. readlib 192
9.1.92. readstat 193
9.1.93. rem and quo 193
9.1.94. Rem and Quo 193
9.1.95. resultant 194
9.1.96. Resultant 195
9.1.97. RETURN 195
9.1.98. RootOf 196
9.1.99. rsolve 196
9.1.100. sign 197
9.1.101. signum 198
9.1.102. simplify 198
9.1.102.1. simplify/atsign 199
9.1.102.2. simplify/GAMMA 199
9.1.102.3. simplify/hypergeom 200
9.1.102.4. simplify/power 200
9.1.102.5. simplify/Psi 200
9.1.102.6. simplify/radical 201
9.1.102.7. simplify/RootOf 201
9.1.102.8. simplify/sqrt 201
9.1.102.9. simplify/trig 202

9.1.103. solve 202
9.1.103.1. solve/float 203
9.1.103.2. solve/function 204
9.1.103.3. solve/identity 204
9.1.103.4. solve/ineq 205
9.1.103.5. solve/linear 205
9.1.103.6. solve/radical 206
9.1.103.7. solve/scalar 206
9.1.103.8. solve/series 207
9.1.103.9. solve/system 207
9.1.104. sort 208
9.1.105. sqrt 209
9.1.106. subs 209
9.1.107. subsop 210
9.1.108. substring 210
9.1.109. sum 211
9.1.110. system 211
9.1.111. table 212
9.1.112. taylor 213
9.1.113. testeq 214
9.1.114. time 214
9.1.115. trace 215
9.1.116. traperror 215
9.1.117. trigonometric functions 216
9.1.118. trunc, round, and frac 216
9.1.119. type 216
9.1.119.1. type/algfunc 217
9.1.119.2. type/algnum 218
9.1.119.3. type/boolean, type/relation and type/logical 218
9.1.119.4. type/constant 218

9.1.119.5. type/expanded 219
9.1.119.6. type/laurent 219
9.1.119.7. type/linear, type/quadratic, type/cubic, type/quartic 220
9.1.119.8. type/listlist 220
9.1.119.9. type/matrix 221
9.1.119.10. type/monomial 221
9.1.119.11. type/point 222
9.1.119.12. type/polynom 222
9.1.119.13. type/radical and type/sqrt 223
9.1.119.14. type/ratpoly 224
9.1.119.15. type/scalar 225
9.1.119.16. type/taylor 225
9.1.119.17. type/trig 225
9.1.119.18. type/vector 226
9.1.120. unames 226
9.1.121. whattype 227
9.1.122. with 227
9.1.123. words 228
9.1.124. writeto 228
9.1.125. Zeta 229
9.2. Miscellaneous library functions 230
9.2.1. bspline 230
9.2.2. C 230
9.2.3. chrem 230
9.2.4. Ci 231
9.2.5. coeftayl 231
9.2.6. commutat 232
9.2.7. compoly 233
9.2.8. convergs 233
9.2.9. D 234

9.2.10. dilog 234
9.2.11. dinterp 235
9.2.12. discrim 235
9.2.13. edit 235
9.2.14. Ei 236
9.2.15. eqn 237
9.2.16. eulermac 237
9.2.17. evalm 238
9.2.18. fibonacci 238
9.2.19. fixdiv 239
9.2.20. forget 239
9.2.21. fortran 240
9.2.22. freeze, thaw 240
9.2.23. hypergeom 241
9.2.24. invlaplace 241
9.2.25. iroot 242
9.2.26. iscont 242
9.2.27. isolate 243
9.2.28. issqr 243
9.2.29. laplace 244
9.2.30. latex 244
9.2.31. match 245
9.2.32. minimize and maximize 246
9.2.33. mtaylor 246
9.2.34. optimize 247
9.2.35. permute 248
9.2.36. priqueue 248
9.2.37. profile, unprofile, showprofile 249
9.2.38. psqrt and proot 250
9.2.39. randpoly 250
9.2.40. realroot 251

9.2.41. replot 252
9.2.42. residue 252
9.2.43. S 253
9.2.44. search 253
9.2.45. select 253
9.2.46. showtime 254
9.2.47. Si 254
9.2.48. singular 255
9.2.49. sinterp 255
9.2.50. stir1 and stir2 256
9.2.51. symmpoly 256
9.2.52. totorder 257
9.2.53. translate 257
9.2.54. trigsubs 258
9.2.55. unapply 258
9.2.56. write, writeln, open, close 259
9.2.57. zip 259
9.3. The `difforms` package for Differential Forms 260
9.3.1. type/const 260
9.3.2. d 261
9.3.3. defform 261
9.3.4. type/form 262
9.3.5. formpart 263
9.3.6. mixpar 263
9.3.7. parity 263
9.3.8. type/scalar 264
9.3.9. scalarpart 264
9.3.10. simpform 265
9.3.11. wdegree 265
9.3.12. &^ 265

9.4. The `grobner` package for Grobner Bases 267
9.4.1. gbasis 267
9.4.2. leadmon 268
9.4.3. normalf 269
9.4.4. spoly 270
9.5. The `group` package for permutation and finitely-presented groups 271
9.5.1. centralizer 271
9.5.2. convert/disjcyc 272
9.5.3. cosets 272
9.5.4. cosrep 273
9.5.5. grelgroup 274
9.5.6. groupmember 274
9.5.7. grouporder 275
9.5.8. inter 275
9.5.9. invperm 275
9.5.10. isnormal 276
9.5.11. mulperms 276
9.5.12. normalizer 276
9.5.13. permgroup 277
9.5.14. permrep 278
9.5.15. pres 278
9.5.16. subgrel 279
9.5.17. type/disjcyc 279
9.6. The `linalg` package for Linear Algebra 281
9.6.1. add 281
9.6.2. addcol 282
9.6.3. addrow 282
9.6.4. adjoint 282
9.6.5. angle 283
9.6.6. band 283
9.6.7. bezout 283

9.6.8. coldim 284
9.6.9. colspace or range 284
9.6.10. cond 284
9.6.11. crossprod 285
9.6.12. curl 285
9.6.13. definite 286
9.6.14. det 286
9.6.15. diverge 287
9.6.16. dotprod 287
9.6.17. eigenvals 287
9.6.18. gausselim 288
9.6.19. genmatrix 288
9.6.20. grad 289
9.6.21. hadamard 289
9.6.22. hessian 289
9.6.23. hilbert 290
9.6.24. indexfunc 290
9.6.25. inverse 290
9.6.26. ismith 291
9.6.27. jacobian 291
9.6.28. kernel or nullspace 292
9.6.29. laplacian 292
9.6.30. leastsqrs 292
9.6.31. linsolve 293
9.6.32. mdet 293
9.6.33. mulcol 294
9.6.34. mulrow 294
9.6.35. multiply 294
9.6.36. norm 295
9.6.37. orthog 295
9.6.38. permanent 296

9.6.39. rank 296
9.6.40. rowdim 296
9.6.41. rowspace 296
9.6.42. scalarmul 297
9.6.43. singularvals 297
9.6.44. smith 297
9.6.45. submatrix 298
9.6.46. subvector 298
9.6.47. swapcol 299
9.6.48. swaprow 299
9.6.49. sylvester 299
9.6.50. toeplitz 300
9.6.51. trace 300
9.6.52. transpose 300
9.6.53. vandermonde 301
9.6.54. vectdim 301
9.7. The `np` package for the Newman-Penrose Formalism 302
9.7.1. Newman-Penrose Pfaffian operators D, V, X, Y 302
9.7.2. Newman-Penrose commutators V_D, X_D, Y_D, X_V, Y_V, Y_X 303
9.7.3. conj 303
9.7.4. eqns 303
9.7.5. suball 304
9.8. The **`numtheory`** package for Number Theory 305
9.8.1. F 305
9.8.2. M 305
9.8.3. cfrac 306
9.8.4. cyclotomic 306
9.8.5. divisors 306
9.8.6. factorset 307

9.8.7. imagunit 307
9.8.8. isqrfree 307
9.8.9. jacobi 307
9.8.10. lambda 308
9.8.11. legendre 308
9.8.12. mcombine 308
9.8.13. mlog 309
9.8.14. mobius 309
9.8.15. mroot 309
9.8.16. msqrt 310
9.8.17. phi 310
9.8.18. pprimroot 310
9.8.19. primroot 311
9.8.20. rootsunity 311
9.8.21. safeprime 312
9.8.22. tau 312
9.9. The `orthopoly` package for Orthogonal Polynomials 313
9.9.1. G 313
9.9.2. H 314
9.9.3. L 314
9.9.4. P 315
9.9.5. T 316
9.9.6. U 316
9.10. The `powseries` package for Formal Power Series 317
9.10.1. add 317
9.10.2. compose 318
9.10.3. evalpow 318
9.10.4. inverse 319
9.10.5. multconst 319

9.10.6. multiply 319
9.10.7. negative 320
9.10.8. powcreate 320
9.10.9. powdiff 321
9.10.10. powexp 321
9.10.11. powint 321
9.10.12. powlog 322
9.10.13. powpoly 322
9.10.14. powsolve 323
9.10.15. quotient 323
9.10.16. representation 324
9.10.17. reversion 324
9.10.18. subtract 325
9.10.19. tpsform 325
9.11. The `simplex` package for Linear Optimization 327
9.11.1. basis 327
9.11.2. convexhull 328
9.11.3. cterm 328
9.11.4. dual 328
9.11.5. equality 329
9.11.6. feasible 329
9.11.7. maximize 330
9.11.8. minimize 330
9.11.9. nonneg 331
9.11.10. pivot 331
9.11.11. pivoteqn 332
9.11.12. pivotvar 332
9.11.13. ratio 333
9.11.14. setup 333
9.11.15. std 334

9.11.16. standardize 334
9.11.17. stdle 334
9.11.18. variables 335
9.12. The `stats` package for Statistics 336
9.12.1. ChiSquare 336
9.12.2. Exponential 337
9.12.3. Fdist 337
9.12.4. Ftest 337
9.12.5. N 338
9.12.6. Rsquared 338
9.12.7. StudentsT 339
9.12.8. Uniform 339
9.12.9. addrecord 339
9.12.10. average 340
9.12.11. correlation 340
9.12.12. covariance 340
9.12.13. evalstat 341
9.12.14. getkey 341
9.12.15. linregress 341
9.12.16. median 342
9.12.17. mode 342
9.12.18. multregress 343
9.12.19. projection 343
9.12.20. putkey 344
9.12.21. regression 344
9.12.22. removekey 345
9.12.23. sdev 345
9.12.24. serr 345
9.12.25. statplot 346
9.12.26. variance 346

9.13. The `student` package for Calculus 348
9.13.1. D 348
9.13.2. Eval 349
9.13.3. Int 349
9.13.4. Limit 350
9.13.5. type/Point 350
9.13.6. Sum 351
9.13.7. changevar 351
9.13.8. cmpltsq 352
9.13.9. combine 352
9.13.10. diff 353
9.13.11. distance 353
9.13.12. intercept 354
9.13.13. intparts 354
9.13.14. isolate 354
9.13.15. makeproc 355
9.13.16. midpoint 356
9.13.17. powsubs 356
9.13.18. slope 357
10. MISCELLANEOUS FACILITIES 358
10.1. Debugging facilities: Detecting syntax errors 358
10.1.1. `Mint`, the Maple program diagnostician 358
10.2. Debugging facilities: Monitoring run-time execution 361
10.2.1. The `printlevel` facility 361
10.2.2. Tracing specific procedures 363
10.3. Monitoring space and time 363
10.3.1. Run-time messages 363
10.3.2. The `status` variable 364
10.3.3. System functions for space and time 365
10.4. Global variables 365

10.5. Maple command line options 367
10.5.1. The Quiet option 367
10.5.2. The Library Specification option 367
10.5.3. The Suppress Initialization option 367
10.6. Other facilities 368
10.6.1. Escape to host 368
Appendix A – Maple under the DEC VMS operating system 369
A.1. Maple and VMS 369
A.2. Maple initialization files 369
A.3. Interrupt and Cancel characters 370
A.4. Temporarily escaping from Maple 370
A.5. Translating between Maple file names and VMS file names 370
A.6. Redirection of input and output 371
A.7. MAPLE command line options for VMS 373
A.7.1. Overview of Maple command line options 373
A.7.2. Library specification option: -b 373
A.7.3. Suppress initialization option: -s 373
A.7.4. Quiet option: -q 373
A.8. MINT 374
A.9. Summary of site- and VMS- dependent aspects of Maple 374
Appendix B – Maple under the IBM VM/CMS operating system 377
B.1. Maple and CMS 377
B.2. Maple initialization files 377
B.3. Temporarily escaping from Maple 377
B.4. Translating between Maple file names and CMS file names 378
B.4.1. Translation rules 378

B.4.2. Examples 380
B.4.3. Convert utility 380
B.5. Redirection of input and output 381
B.6. MAPLE command line options for CMS 381
B.6.1. Overview of Maple command line options 381
B.6.2. Suppress initialization option: SUPPRESS 382
B.6.3. Quiet option: QUIET 382
B.7. Prompt 382
B.8. MINT 382
B.9. Summary of site- and CMS- dependent aspects of Maple 384
Appendix C – Maple under the UNIX operating system 386
C.1. Maple and Unix 386
C.2. Maple initialization files 386
C.3. Quit and Interrupt characters 386
C.4. Temporarily escaping from Maple 387
C.5. Redirection of input and output 387
C.6. `maple` command line options for Unix 388
C.6.1. Overview of Maple command line options 388
C.6.2. Library specification option: -b 388
C.6.3. Suppress initialization option: -s 388
C.6.4. Quiet option: -q 388
C.7. `mint` 389
C.8. Summary of site- and Unix- dependent aspects of Maple 390
References 392
Index 393

1. INTRODUCTION

Maple is a mathematical manipulation language. (The name can be said to be derived from some combination of the letters in the preceding phrase, but in fact it was simply chosen as a name with a Canadian identity.) The type of computation provided by Maple is known by various other names such as 'algebraic manipulation', 'symbolic computation', or 'computer algebra'. A basic feature of such a language is the ability to, explicitly or implicitly, leave the elements of a computation unevaluated. A corresponding feature is the ability to perform 'simplification' of expressions involving unevaluated elements.

In top-level Maple, statements are normally evaluated as far as possible in the current 'environment'. For example the statement

```
a := 1;
```

assigns the value 1 to the name a. If this statement is later followed by the statement

```
x := a + b;
```

then the value 1+b is assigned to the name x. Next if the assignments

```
b := -1; f := sin(x);
```

are performed then x evaluates to 0 and the value 0 is assigned to the name f. (Note that sin(0) is automatically 'simplified' to 0). Finally if we now perform the assignments

```
b := 0; g := sin(x);
```

then x evaluates to 1 and the value sin(1) is assigned to the name g. (Note that sin(1) cannot be further evaluated or simplified in a symbolic context, but there is a facility to evaluate an expression into floating-point form which will yield the decimal expansion of sin(1) to a number of digits controlled by the user.)

As each complete statement is entered by a user, it is evaluated and the results are printed on the output device, usually the user's terminal. The printing of expressions is normally presented in a two-dimensional, multi-line format designed to be as readable as possible on standard computer terminals. There are several ways in which the user can alter the display of expressions. If the global variable 'prettyprint' is assigned the value 0 then expressions are printed in a one-dimensional line-printing mode that is more

suitable for machine input than for human readability. The default value of 'prettyprint' is 1, resulting in a two-dimensional display which is centered in the case of a single-line expression and which breaks the expression onto several lines of two-dimensional output as required. Assigning 'prettyprint' the value 2 results in a left-justified two-dimensional display of expressions.

The global variable 'printlevel' determines how much output will be printed. Setting this variable to -1 prevents the printing of any results, which may be desired to silently read a file of Maple statements, for example. The default value of 'printlevel' is 1, and in general, higher integer values cause more intermediate steps in a computation to be displayed. (See chapter 10 for more details.) Finally, the silent statement separator ':', when used to terminate a statement at the interactive level, prevents the results of the preceding statement from being displayed. (The normal statement separator is ';'.)

There is an on-line **help** facility in the Maple system. To start using it, enter the statement

```
help();
```

and some information about the help facility will be displayed at the terminal. Examples of particular calls to the help function are:

```
help(index);  help(gcd);  help(`if`);  help(linalg, eigenvals);
```

2. LANGUAGE ELEMENTS

2.1. Character Set

The Maple character set consists of letters, digits, and special characters. The letters are the 26 *lower case letters*

```
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
```

and the 26 *upper case letters*

```
A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z.
```

The 10 *digits* are

```
0,1,2,3,4,5,6,7,8,9
```

and the 30 *special characters* are

	blank	[	left bracket
;	semicolon	]	right bracket
:	colon	{	left brace
+	plus	}	right brace
–	minus	`	grave accent (back quote)
*	asterisk	'	apostrophe (single quote)
/	slash	"	double quote (ditto)
^	caret (circumflex)	=	equal
!	exclamation	<	less than (left angle bracket)
@	at sign	>	greater than (right angle bracket)
$	dollar sign	|	vertical bar
.	period	&	ampersand
,	comma	_	underscore
(	left parenthesis	\	backslash
)	right parenthesis	#	sharp

There are two-character alternates for five of these special characters, for use when the complete ASCII character set is not available. The five characters and their alternates are:

^	caret (circumflex)	**
[	left bracket	(|
]	right bracket	|)
{	left brace	(*
}	right brace	*)

2.2. Tokens

The tokens consist of keywords, programming-language operators, strings, natural integers, and punctuation marks. The *keywords* are the following 23 reserved words:

by	do	done	elif
else	end	fi	for
from	if	in	local
od	option	options	proc
quit	read	save	stop
then	to	while	

There are three types of *programming-language operators*, namely the *binary*, *unary*, and *nullary* operators. The *binary* operators are:

Operator	Meaning	Operator	Meaning
+	addition	<	less than
–	subtraction	<=	less or equal
*	multiplication	>	greater than
/	division	>=	greater or equal
**	exponentiation	=	equal
^	exponentiation	<>	not equal
$	sequence operator	`mod`	modulo
@	composition	`union`	set union
@@	repeated composition	`minus`	set difference
&string	neutral operator	`intersect`	set intersection
.	concatenation; decimal	`and`	logical and
..	ellipsis (in general, ...†)	`or`	logical or
,	expression separator	:=	assignment

† Two *or more* consecutive periods are parsed as an ellipsis.

The *unary* operators are:

Operator	Meaning
+	unary plus (prefix)
−	unary minus (prefix)
!	factorial (postfix)
$	sequence operator (prefix)
`not`	logical not (prefix)
&string	neutral operator (prefix)
.	decimal point (prefix or postfix)

The *nullary* operators are:

Operator	Meaning
%	latest expression
%%	penultimate expression
%%%	before penultimate expression

(Note that seven of these operators are reserved words: `mod`, `union`, `minus`, `intersect`, `and`, `or`, `not`.)

The simplest instance of a *string* is a letter followed by zero or more letters, digits, and underscores. Another instance of a string is the underscore followed by zero or more letters, digits, and underscores. More generally, a string can be formed by enclosing any sequence of characters in back quotes (grave accents). In all cases, the maximum length of a string in Maple is 495 characters. A string is a valid name (e.g., a variable name or a function name) but the user should avoid using names which begin with an underscore since such names are created and used globally by the Maple system. A Maple string is also used in the sense of a 'character string', usually by enclosing a sequence of characters in back quotes. (See section 3.2.2 for more details and examples on string and name formation, including the issue of enclosing special characters such as back quote, backslash, or newline within a string.)

A *natural integer* is any sequence of one or more digits. The numeric constants in Maple (integers, rational numbers, and floating-point numbers) are formed from the natural integers using programming-language operators. The length of a natural integer (and hence the length of integers, rational

numbers, and floating-point numbers) is arbitrary (i.e., the length limit is system dependent but generally much larger than users will encounter).

The *punctuation marks* are:

;	semicolon	(	left parenthesis
:	colon	)	right parenthesis
'	single quote	[	left bracket
`	back quote	]	right bracket
|	vertical bar	{	left brace
<	left angle bracket	}	right brace
>	right angle bracket		

The semicolon and the colon are used to separate statements. The distinction is that during an interactive session if a statement is followed by a colon, the result of the statement is not printed. Enclosing an expression in a pair of single quotes specifies that the expression is to be left unevaluated. The back quote is used to form strings. The left and right parentheses are used to group terms in an expression and to group parameters in a function call. The left and right brackets are used to form indexed (subscripted) names and to select components from aggregate objects such as arrays, tables, and lists. The left and right brackets are also used to form lists and the left and right braces are used to form sets. The left and right angle brackets are used to form functional operators, as is the vertical bar which serves to separate the body of the functional operator from the formal parameters and local variables.

2.3. Blanks, Lines, Comments, and Continuation

The *white space* characters are ⟦space⟧, ⟦tab⟧, ⟦carriage return⟧, and ⟦line feed⟧. We will use the terminology *newline* to refer to either ⟦carriage return⟧ or ⟦line feed⟧ since these characters are not distinguished in the Maple system. Whenever the terminology *blank* is used, it is understood to mean either ⟦space⟧ or ⟦tab⟧. The white space characters separate tokens, but are not themselves tokens. White space characters cannot occur within a token but otherwise may be used freely. The one exception to the free use of white space characters is the formation of a string by enclosing a sequence of characters within back quotes, in which case the white space characters are significant like any other character.

Input to the Maple system consists of a sequence of statements separated by the statement separators, which are the semicolon and the colon characters. The system operates in an interactive mode, executing statements as they are entered. A *line* consists of a sequence of characters followed by a newline character. A single line may contain several statements or it may contain an incomplete statement (i.e., a statement to be completed on succeeding lines), or it may contain several statements followed by an incomplete statement. When a line is entered, the system evaluates (executes) the statements (if any) which have been completed on that line. At the interactive level, a statement will be recognized as complete only when a statement separator is encountered. (One exception is the **quit** statement which need not be followed by a statement separator.)

On a line, all characters which follow a sharp character (#) are considered to be part of a *comment*. The comment is echoed by the system. However, the sharp character is not treated as the beginning of a comment if it is enclosed within a pair of back quotes (in which case it is a character within a string).

Since a newline character may be used freely anywhere except within a token, statements may be continued from line to line quite naturally. The problem of continuation from one line to the next is less trivial when long numbers or long strings are involved, since these are the two classes of tokens which are not restricted to being only a few characters in length. The general mechanism in Maple to specify continuation of one line onto the next line is as follows: if a newline character is immediately preceded by the special character backslash (\ ) then both the backslash and the newline are ignored by the parser. Another property of the backslash character is that it is itself ignored if it is immediately followed by any character other than newline. This is useful when it is desired to break up a long sequence of digits into groups of smaller sequences to enhance readability. (See section 3.2.1 for examples of specifying long numbers using grouping of digits, and using continuation onto a new line.)

For the case of strings formed as a sequence of characters enclosed within back quotes, the blank and newline characters are all valid characters within a string (although, as a safety feature, a warning message is generated whenever a newline is encountered within a string). If it is desired to make the newline transparent in the specification of a string (i.e., to allow continuation of the string specification onto a new line rather than the insertion of a newline character into the string) then the general mechanism discussed

above must be used: the newline must be immediately preceded by the special character backslash (\ ). The backslash character is itself normally ignored, as noted above, but if two backslash characters occur in succession then only the first one is ignored, yielding a mechanism for specifying backslash as one of the characters in a string. (See section 3.2.2 for examples of string specification.)

2.4. Files

The file system is an important part of the Maple system. The user interacts with the file system either explicitly by way of the **read** and **save** statements, or implicitly by specifying a function name corresponding to a file which the system will read in automatically. The file naming conventions used within the Maple system do not necessarily correspond directly with the file naming conventions on the host operating system. A translation from Maple's filenames to the host system's filenames takes place automatically. The **convert** function may be used in the form

convert(⟦Maple filename⟧, hostfile)

to cause Maple to display the actual filename that will be used on the host system. (See the appendices for further details.)

A *file* consists of a sequence of statements either in 'Maple internal format' or in 'user format'. If the file is in user format then the effect of reading the file is identical to the effect of the user entering the same sequence of statements. The system will display the result of executing each statement which is read in from the file. On the other hand, if the file is in Maple internal format then reading the file causes no information to be displayed to the user but updates the current Maple environment with the contents of the file. Maple assumes that a file will be in Maple internal format when its filename ends with the characters '.m' . For example, some typical names for files in user format are:

temp
`lib/src/gcd`

while some typical names for files in internal format are:

```
`temp.m`
`lib/gcd.m`
```

(Note that filenames involving characters such as '/' or '.' must be enclosed in back quotes in order to be interpreted properly as [[name]]s.)

There are two ways to create a file in user format. One way is to use a text editor on the host system to create a file in which Maple statements have been entered just as they would be entered in an interactive Maple session. Another way is to use the Maple **save** statement from within a Maple session, saving into a file whose name does not have the '.m' suffix. The latter action writes the values of specified variables (or all assigned variables if none are specified) into the file as user-readable Maple assignment statements. A file in Maple internal format can be created only by using the **save** statement from within a Maple session, saving into a file whose name has the suffix '.m'. Internal-format files are not in a form that can be read normally; they are meant to be read only by the Maple system. Either type of file may be read (loaded) into a Maple session by using the Maple **read** statement. Many Maple functions are not part of the basic Maple system which is loaded in initially, but rather reside in files in the Maple library. When one of these functions is invoked in Maple, the internal-format file containing the function definition is automatically loaded. (See chapter 9 – The Maple Library.)

3. STATEMENTS AND EXPRESSIONS

3.1. Types of Statements

There are eight types of statements in Maple. They will be described informally here. The formal syntax is given in section 4.2.

3.1.1. Assignment Statement

The form of this statement is

[[name]] := [[expression]]

and it associates a name with the value of an expression.

3.1.2. Expression

An [[expression]] is itself a valid statement. The effect of this statement is that the expression is evaluated.

3.1.3. Read Statement

The statement

read [[expression]]

causes a file to be read into the Maple session. The [[expression]] must evaluate to a name which specifies the file. The file name may be one of two types as discussed in section 2.4. A typical example of a read statement is

read `lib/f.m`

where the back quotes are necessary so that the expression evaluates to a name.

3.1.4. Save Statement

The save statement has a form corresponding to the read statement:

save [[expression]]

and it can also be used in the more general form:

save ⟦nameseq⟧, ⟦expression⟧ .

The simpler form causes the current Maple environment to be written into a file. The ⟦expression⟧ must evaluate to a name which specifies the file. If the file name ends with the characters '.m' then the environment is saved in Maple internal format, otherwise the environment is saved in user format as a sequence of assignment statements.

The generalized syntax for the save statement allows the specification of a sequence of specific names to be saved, yielding a *selective save* facility. It takes the general form

save name1, name2, . . . , namek, filename

where name1, . . . , namek are the names of assigned variables to be saved and filename evaluates to the name of a file. Each argument except the last one is "evaluated to a name" while the last argument is fully evaluated. The construct ⟦name⟧ . (⟦range⟧) may be used here to construct a sequence of names. (See section 3.2.8.)

3.1.5. Selection Statement

The selection statement takes one of the following general forms. Here ⟦expr⟧ is an abbreviation for ⟦expression⟧ and ⟦statseq⟧ stands for a sequence of statements.

if ⟦expr⟧ then ⟦statseq⟧ fi
if ⟦expr⟧ then ⟦statseq⟧ else ⟦statseq⟧ fi
if ⟦expr⟧ then ⟦statseq⟧ elif ⟦expr⟧ then ⟦statseq⟧ fi
if ⟦expr⟧ then ⟦statseq⟧ elif ⟦expr⟧ then ⟦statseq⟧ else ⟦statseq⟧ fi

Wherever the construct 'elif ⟦expr⟧ then ⟦statseq⟧' appears in the above forms, this construct may be repeated any number of times to yield a valid selection statement. The sequence of statements in the branch selected (if any) is executed. The construct 'elif ⟦expr⟧ then ⟦statseq⟧' has the meaning 'else if ⟦expr⟧ then ⟦statseq⟧', but the latter form would require a closing 'fi' delimiter for each opening 'if'. Use of the 'elif' keyword avoids the need for multiple 'fi' delimiters. The selection statement can be viewed as a form of a 'case statement' since multiple conditions can be specified, and clearly the 'else' clause is the 'default case'.

3.1.6. Repetition Statement

The syntax of the repetition statement is either of the following two forms, where ⟦expr⟧ and ⟦statseq⟧ are as above:

for ⟦name⟧ from ⟦expr⟧ by ⟦expr⟧ to ⟦expr⟧ while ⟦expr⟧ do ⟦statseq⟧ od

for ⟦name⟧ in ⟦expr⟧ while ⟦expr⟧ do ⟦statseq⟧ od

where any of 'for ⟦name⟧', 'from ⟦expr⟧', 'by ⟦expr⟧', 'to ⟦expr⟧', or 'while ⟦expr⟧' may be omitted. The sequence of statements in ⟦statseq⟧ is executed zero or more times. The 'for ⟦name⟧' part may be omitted if the index of iteration is not required in the loop, in which case a 'dummy index' is used by the system.

In the first form above, if the 'from ⟦expr⟧' part or the 'by ⟦expr⟧' part is omitted then the default value 'from 1' or 'by 1', respectively, is used. If the 'to ⟦expr⟧' part or the 'while ⟦expr⟧' part is present then the corresponding tests for termination are checked at the beginning of each iteration, and if neither is present then the loop will be an infinite loop unless it is terminated by the **break** construct (see below), or by a **return** from a procedure (see section 6.7), or by execution of the **quit** statement (see section 3.1.8).

In the second form of the repetition statement, the index (i.e., the ⟦name⟧ specified in the for-part of the statement) takes on, in order, values from the list [op(⟦expr⟧)] where ⟦expr⟧ is the expression specified in the 'in ⟦expr⟧' part of the statement (or if ⟦expr⟧ is itself an *expression sequence* then the index simply takes on values from that sequence). In other words, the successive values of the index are the successive operands of ⟦expr⟧. (See section 4.1 for a description of the operands associated with each Maple data type.) The value of the index may be tested in the while-part of the statement and, of course, the value of the index is available when executing the ⟦statseq⟧. As in the first form, if the 'while ⟦expr⟧' part is present then the test for termination is checked at the beginning of each iteration.

There are two additional loop control constructs in the Maple language: **break** and **next**. When the special name 'break' is evaluated, the result is to exit from the innermost repetition statement within which it occurs. Execution then proceeds with the first statement following this repetition statement. When the special name 'next' is evaluated, the result is to exit from the current ⟦statseq⟧ corresponding to the innermost repetition statement within which it occurs, and to cause execution to proceed with the next iteration of this repetition statement. Note that "to proceed with the next iteration"

implies updating the index of iteration and then applying the tests for termination (if any) before proceeding, so an exit from the repetition statement will occur if the tests for termination so indicate. It is an error if the names 'break' or 'next' are evaluated in a context other than within a repetition statement.

3.1.7. Empty Statement

The empty statement is syntactically valid in Maple. For example

```
a := 1; ; quit
```

is a valid statement sequence in Maple consisting of an assignment statement, the empty statement, and the quit statement. Of course since blanks may be freely used, any number (including zero) of blanks could appear between the semicolons here yielding a syntactically identical statement sequence.

3.1.8. Quit Statement

The syntax of the quit statement is any one of the following three forms:

```
quit
done
stop
```

The result of this statement is to terminate the Maple session and return the user to the system level from which Maple was entered. (In many implementations of Maple, pressing the break/interrupt key twice in rapid succession, or some similar system-specific sequence, can also be used to cause termination of a Maple session – see the appendices for system-specific details.)

3.2. Expressions

Expressions are the fundamental entities in the Maple language. The various types of expressions are described informally here. The formal syntax is given in section 4.2.

3.2.1. Constants

The numeric constants in Maple are integers, fractions, and floating-point numbers. A ⟦natural integer⟧ is any sequence of one or more digits of arbitrary length (i.e., the length limit is system dependent but generally much larger than users will encounter). An *integer* is a ⟦natural integer⟧ or a signed integer (i.e. +⟦natural integer⟧ or −⟦natural integer⟧). A *fraction* is of the form

⟦integer⟧ / ⟦natural integer⟧

(Note that a fraction is always simplified by Maple so that the denominator is positive, and it will also be reduced to lowest terms).

An ⟦unsigned float⟧ is one of the following three forms:

⟦natural integer⟧ . ⟦natural integer⟧
⟦natural integer⟧ .
. ⟦natural integer⟧

A *floating-point number* is an ⟦unsigned float⟧ or a signed float (i.e. +⟦unsigned float⟧ or −⟦unsigned float⟧). The **evalf** function is used to force an expression to be evaluated to a floating-point number (if possible). The number of digits carried in the 'mantissa' when evaluating floating-point numbers is determined by the value of the global name 'Digits' which has 10 as its initial value.

It is sometimes necessary to specify a numeric constant whose length exceeds the length of a single line. Since a newline character is not allowed within a ⟦natural integer⟧ token, Maple's general continuation mechanism (see section 2.3) must be used when line-breaking within a sequence of digits. Namely, the newline character must be immediately preceded by the special character backslash (\ ), in which case both the backslash and the newline character will be ignored. Some examples are given below. Since another property of the backslash character is that it is itself ignored if it is immediately followed by any character other than newline, the following examples also include cases where we have chosen to group the digits into groups of three to enhance readability. For example, the following two forms of input yield the same floating-point number:

```
3.141592653589793
3.141\592\653\589\793
```

and two additional forms of input yielding this same number are:

```
3.
141592653589793

3.141592653\
589793
```

In the four forms of input above, the first form uses no line-breaking and no grouping of digits, the second form uses the transparent backslash character to break the sequence of digits into groups of three, the third form breaks the line at the decimal point in which case it is not necessary to precede the newline character with a backslash character, and the fourth form uses a line-break in the middle of a sequence of digits in which case it is necessary to precede the newline with a backslash.

An alternative input format for floating-point numbers in Maple is to use the notation Float(mantissa, exponent) which directly corresponds to Maple's internal data structure representation of floating-point numbers. In this notation, both arguments to the Float function must be integers and the value is interpreted to be the floating-point number

$$\text{mantissa} \times 10^{\text{exponent}}.$$

For example, Float(3141592654, −9) represents the same floating-point number as the more standard input notation 3.141592654, and it is also the value of any of the following input expressions:

```
3141592654. * 10^(-9)
.3141592654 * 10^1
314.1592654 * 10^(-2)
```

When Maple's line-printing mode is used (via the **lprint** function, or in normal display when pretty-printing is turned off by assigning zero to the global name *prettyprint*), the Float(mantissa, exponent) notation is used for floating-point numbers with very small or very large magnitudes. For example,

```
prettyprint := 0: evalf(exp(-10)); evalf(exp(-20));
```

yields

```
Float(4539992976,-14)
Float(2061153622,-18)
```

in contrast with the normal display of these numbers which would be seen in

pretty-printed format as:

$$.00004539992976$$

$$.2061153622*10^{-8}$$

There is the concept of *symbolic constants* in Maple. The global variable 'constants' is initially assigned the following expression sequence of names:

false, gamma, infinity, true, Catalan, E, I, Pi

implying that these particular names are understood by Maple to be of type *constant*. The user may define additional names (specifically, they must be the simplest type of names called *strings* – see section 3.2.2) to be constants by redefining the value of this global variable. For example, if the following assignment is executed:

constants := constants, g;

then the name "g" has been added to the constants sequence and type(g,constant) returns *true*. More generally, a Maple expression is of type *constant* if it is an unevaluated function with all arguments of type *constant*, or a sum, product, or power with all operands of type *constant*. For example, the following expressions are of type *constant*:

sin(1), f(2,3), exp(gamma), 4 + Pi, 2*Pi*I .

3.2.2. Names and Strings

A name in Maple has a value which may be any expression or, if no value has been assigned to it, then it stands for itself. A name is usually a ⟦string⟧, which in its simplest form is a letter followed by zero or more letters, digits, and underscores (with a maximum length of 495 characters). Note that there are 30 reserved words in the Maple system (23 keywords plus 7 programming-language operators) which are not generally available as names formed in the above manner. Note also that lower case letters and upper case letters are distinct, so that the names

g G new_term New_Term x13a x13A

are all distinct. Another instance of a ⟦string⟧ is the underscore followed by zero or more letters, digits, and underscores. Names beginning with the underscore are used as global variable names by the Maple system and therefore such names should be avoided by users.

A ⟦string⟧ can also be formed by enclosing any sequence of characters in back quotes. (To allow the back quote to be one of the characters in a string, two consecutive back quotes appearing after the opening of a string are parsed as an enclosed back quote character.) A reserved word enclosed in back quotes also becomes a valid Maple string, distinct from the usage of the reserved word as a token. The following are valid strings (and hence names) in Maple:

```
`This is a strange name:`   `2D`   `n-1`   `a``b`   `while` .
```

The back quotes do not themselves form part of the string so they disappear when the string has been input to Maple. For example, if the fourth string in the above list is input to Maple, the response will be the string

```
a`b .
```

As another example, if n has the value 5 then the statement

```
`n-1` := n-1;
```

will yield the following response from Maple:

```
n-1 := 4 .
```

The user should beware of misusing this facility for string (name) formation to the point of writing unreadable programs!

The effect of a newline character within a string should be noted. A newline character is a valid character within a string. However, a warning message is issued whenever a newline occurs within a string. This warning message is a safety precaution to help the user identify the following type of bug. A common typographical mistake is to fail to close a string which has been opened, in which case all succeeding input lines are 'eaten up' as part of the string. In this situation, the user finds that no matter what command is typed in the Maple session, there is no response from Maple except to prompt for more input, even when the **quit** statement is entered in an attempt to leave the Maple session out of frustration. With the warning message in place, every line of input yields the response:

```
Warning: String contains newline character. Close strings with ` quote.
```

For example, if the input to Maple is

```
prettyprint := 0:
s := `This is a
string`;
```

where the character following the letter 'a' is a newline ([[carriage return]]), then the response from Maple is

```
Warning: String contains newline character. Close strings with ` quote.
s := This is a
string
```

In this example, prettyprinting was turned off; with prettyprinting turned on, the line containing the assignment operator would be centered on the line. Note that the string assigned to 's' is 16 characters in length, one of the characters being the newline character.

Alternatively, it may be desired to specify a string not containing a newline character, but more than one input line is needed to specify the string. In this case, preceding the newline with a backslash will make the newline transparent, just as in the case of long numbers. For example, consider the statement:

```
if not type(n,integer) or n <= 0 then print(`The value of n is not a \
positive integer.`) fi;
```

If 'n' has the value –2 when this statement is executed, then the result will be to print the string:

```
The value of n is not a positive integer.
```

In this case, the string does not contain the newline character. Finally, it should be noted that the backslash character can be entered as a character in a string by exploiting the fact that two successive backslash characters are parsed as a single backslash character (just as two successive back quotes are parsed as a single back quote). For example, the input

```
`Don't confuse the quotes: ', ", and `` , nor \
the slashes: / and \\ . ` ;
```

yields the following string as a response:

```
Don't confuse the quotes: ', ", and ` , nor the slashes: / and \ .
```

More generally, a [[name]] may be formed using the *concatenation operator* in one of the following three forms:

⟦name⟧ . ⟦natural integer⟧
⟦name⟧ . ⟦string⟧
⟦name⟧ . (⟦expression⟧)

Some examples of the use of the concatenation operator for ⟦name⟧ formation are:

v.5 p.n a.(2*i) V.(N.(i–1)) r.i.j .

The concatenation operator is a binary operator which requires a ⟦name⟧ as its left operand. Its right operand is evaluated and then concatenated to the left operand. If the right operand is evaluated to an integer or a string, then the result of the concatenation is a name. If, on the other hand, the right operand is evaluated to some other type of object, say a floating-point number, then the result of the operation is an unevaluated concatenated object. (c.f. *Unevaluated concatenation* in section 4.1.16.) For example if n has the value 4 then p.n evaluates to the name p4, while if n has no value then p.n evaluates to the name pn. If n has the value 3.0, then p.n does not produce a name, but instead yields an unevaluated concatenated object. If i has the value 5 then a.(2*i) evaluates to the name a10. As a final example, if N4 has the value 17 and i has the value 5 then V.(N.(i–1)) evaluates to the name V17, while V.N.(i–1) evaluates to the name VN4, assuming that N has no value. (See also the function **cat** in section 9.1, for string concatenation.)

Yet another form of name in Maple is the *indexed name* which has the form

⟦name⟧ [⟦expression sequence⟧] .

Note that since an indexed name is itself a valid name, there can be a succession of subscripts as in:

A[1,2,3][x,y][2,1] .

The use of the indexed name b[1] does not imply that 'b' is an array, as is true in many other languages. The statement

a := b[1] + b[2] + b[1000];

simply forms the sum of three indexed names. It is not necessary that 'b' have an array value. (However, if 'b' does evaluate to an array or a table, then 'b[1]' is the element of the array or table selected by '1'.) The assignment of a value to an indexed expression will implicitly create a table. For example, consider the statement

```
b[1] := 10;
```

If 'b' had no value previously, then this assignment statement creates a table for 'b' and then assigns the value 10 to the component of this table indexed by 1. (See chapter 5: Arrays and Tables.)

3.2.3. Expression Sequences

An *expression sequence* is an expression of the form

$$[\![\text{expression}_1]\!] , [\![\text{expression}_2]\!], \ldots , [\![\text{expression}_n]\!]$$

The comma operator is used to join expressions into an expression sequence. It has the lowest precedence of all operators except assignment (see section 3.2.13). When expression sequences are formed, the result is simplified to a single, unnested expression sequence.

A zero-length expression sequence is syntactically valid. It arises, for example, in the context of forming an empty list, an empty set, a function call with no parameters, or an indexed name with no subscripts. The special name 'NULL' is initially assigned the zero-length expression sequence and may be used when needed. Although the name 'NULL' is an ordinary Maple [[name]], it should not be assigned a different value.

The *sequence operator* $ may be used to form expression sequences as follows. The syntax is any of the following general forms:

```
[[expr1]] $ [[name]] = [[expr2]] .. [[expr3]]
[[expr1]] $ [[expr3]]
$ [[expr2]] .. [[expr3]]
```

In the first form, the result is the expression sequence produced by substituting for [[name]] in [[expr1]] the values [[expr2]], [[expr2]] + 1, . . . , [[expr3]] (or up to the last value not exceeding [[expr3]]). If the value [[expr2]] is greater than [[expr3]] then the result is the NULL expression sequence. If the second form is used, it is equivalent to the form

```
[[expr1]] $ i = 1 .. [[expr3]]
```

(assuming that [[expr1]] does not contain i). In other words, the result of the second form is to produce a sequence of length [[expr3]] by repetition of [[expr1]]. If the third form is used, it is equivalent to the form

```
i $ i = [[expr2]] .. [[expr3]]
```

which is a simple mechanism for producing a numeric sequence. In all

cases, if either of ⟦expr2⟧ or ⟦expr3⟧ does not evaluate to a numeric value then the result is an unevaluated function invocation of the function named `$` (but in the usual prettyprinting mode, the output format corresponds to the input format specified above). It should be noted that where a ⟦name⟧ is required in the above forms, it must evaluate to a name; in practice, this may require the use of single quotes to enclose both '⟦expr1⟧' and '⟦name⟧' to prevent unwanted evaluation (assuming that ⟦name⟧ has a previous value).

Examples:

```
p[i] $ i = 1..3;        yields  p[1], p[2], p[3]
i^2 $ i = 2/3 .. 8/3;   yields  4/9, 25/9, 64/9
x$4;                    yields  x, x, x, x
$ 2..5;                 yields  2, 3, 4, 5
a := A, B, C, D;        yields  a := A, B, C, D
b := NULL;              yields  b :=
c := a, b, a;           yields  c := A, B, C, D, A, B, C, D
c[6];                   yields  B
```

The latter statement selects the sixth operand of the expression sequence c (see section 3.2.9).

3.2.4. Sets and Lists

A *set* is an expression of the form

{ ⟦expression sequence⟧ }

and a *list* is an expression of the form

[⟦expression sequence⟧] .

Note that an ⟦expression sequence⟧ may be empty so that the empty set is represented by { } and the empty list is represented by [ ]. A set is an *unordered* sequence of expressions and the user should not assume that the expressions will be maintained in any particular order. (A particular invocation of Maple will use a particular ordering based on the internal addresses of the expressions.) Duplicates are removed from the expression sequence. A list is an *ordered* sequence of expressions with the order of the expressions specified by the user. For example, if the user inputs the set {x, y, y} the system might represent this in the form {y, x} while if the user inputs the list [x, y, y] then the representation used by the system will be precisely this list.

Some operations which are useful for the manipulation of lists are equality testing via the relational operator '=' (see section 3.2.7), the **selection** operation (see section 3.2.9), and the functions **op**, **subsop**, and **map** (see section 9.1).

There are three *set operators* which are used for set arithmetic:

union, minus, intersect .

As the names imply, these operators perform set union, set difference, and set intersection, respectively. As operators, these three names are reserved words in Maple. It is also possible to use the corresponding back-quoted names:

`` `union` , `minus` , `intersect` ``

as function names to achieve the same effect (see Examples below). Some other operations which are useful for the manipulation of sets are equality testing via the relational operator '=' (see section 3.2.7) and the function **member** (see section 9.1) which tests for set membership.

Examples:

```
u := [x,2*x+1,y];          yields  u := [x, 2*x+1, y]
u[3];                      yields  y
v := [ op(2..3, u), exp(x) ];  yields  v := [2*x+1, y, exp(x)]
w := [ u[2..3], exp(x) ];  yields  w := [2*x+1, y, exp(x)]
evalb( v=w );              yields  true
u1 := [ op(u), y^2 ];      yields  u1 := [x, 2*x+1, y, y^2]
u2 := subsop(2=z, u1);     yields  u2 := [x, z, y, y^2]
map(sqrt, u2);             yields  [x^(1/2), z^(1/2), y^(1/2), y]
```

```
s := {x,y} union {y,z};                          yields  s := {x,y,z}
t := `union`({x,y},{y,z});                       yields  t := {x,y,z}
evalb( s=t );                                    yields  true
member( y, s );                                  yields  true
member( y, {x*y, y*z} );                         yields  false
member( x*y, {x*y, y*z} );                       yields  true
{3,4} union a union {3,7} union b union a;       yields  union(a,b,{3,4,7})
{x,y,z} minus {y,z,w};                           yields  {x}
a minus b;                                       yields  a minus b
`minus`(a,a);                                    yields  {}
{x,y,z} intersect {y,z,w};                       yields  {y,z}
`intersect`(a,c,b,a);                            yields  intersect(a,b,c)
```

3.2.5. Algebraic Operators

There are 13 *algebraic operators:*

`%`, `%%`, `%%%`, !, +, –, *, /, **, ^, @, @@, mod .

The nullary operator % has as its value the latest expression, the nullary operator %% has as its value the penultimate expression, and the nullary operator %%% has as its value the expression preceding the penultimate expression. The sequence of expressions defining these three nullary operators is the sequence of expressions generated during a Maple session *at the same level* as the occurrence of the nullary operator, but it specifically excludes the NULL expression sequence and it excludes the values of the index of iteration in a repetition statement. The most common use of these operators is at the top, or interactive, level of a Maple session, in which case the expressions generated by a procedure invocation are "hidden" from these nullary operators because they are at a lower level. These operators may also be used within the body of a procedure, in which case they refer to the sequence of expressions generated during execution of that particular procedure body, excluding expressions generated by any sub-procedures invoked.

The unary operator ! is used as a postfix operator and it denotes the factorial function of its operand. + and – may be used as prefix operators representing unary plus and unary minus. The six operators:

+, –, *, /, **, ^

all may be used as binary operators, representing addition, subtraction,

multiplication, division, exponentiation, and exponentiation, respectively. The two operators ** and ^ are synonymous and may be used interchangeably.

The @ operator is used to represent composition of functional operators. For example:

```
(sin@exp)(x);      yields  sin(exp(x))
(sin@arcsin)(x);   yields  x
```

The @@ operator is the corresponding "exponentiation" operator representing repeated applications of the composition operator. For example:

```
(ln@@2)(x);              yields  ln(ln(x))
((sin@@2)@arcsin)(x);    yields  sin(x)
```

For more detailed information on the algebra of functional operators in Maple, see chapter 7.

The mod operator is used for the operation of evaluating an expression modulo m, for a nonzero integer modulus m. As with the set operators, it is also available as a function with the name `mod`. The representation used for the integers modulo m depends on the specific function assigned to the name `mod`. By default,

```
`mod` := modp
```

which uses the positive representation (i.e., for an integer n the result of the operation r := n mod m lies in the interval $0 \leq r < |m|$). If the user assigns

```
`mod` := mods
```

then the symmetric representation is used (i.e., for an integer n the result of the operation r := n mod m lies in the interval $-\frac{|m|}{2} < r \leq \frac{|m|}{2}$). The functions modp and mods may be invoked directly if desired.

The mod operation understands the inert operator &^ for powering. For example,

```
i &^ j mod m
```

will calculate i^j mod m without first calculating i^j in integer arithmetic. It also understands the function name 'power' for this same purpose:

power(i,j) mod m .

The first operand of the mod operator is not restricted to be an integer expression but rather it may be a general expression. The expression is evaluated over the coefficient field of integers modulo m. In particular, the following functions are known to the mod operator:

Content	Divide	Factor	Gcd	Gcdex
Interp	Prem	Primpart	Quo	Rem
Resultant	RootOf	Sprem		

For further information on these functions see section 9.1.

Examples:

```
`mod`;                              yields  modp
12 mod 7;                           yields  5
7 mod p;                            yields  modp(7,p)
`mod` := mods: 12 mod 7;            yields  -2
`mod` := modp: -1 mod 13;           yields  12
3^(-1) mod 7;                       yields  5
17*x*y^11 + 11*x  mod 11;           yields  6*x*y^11
2 &^ 8 mod 100;                     yields  56
`mod` := mods: 4/3 mod 7;           yields  -1
17*x*y^11 + 11*x  mod 11;           yields  -5*x*y^11
a := 9*x^3 + 9*x^2 + 6*x + 6:
b := 3*x^2 + 8*x + 9:
g := Gcd(a,b) mod 11;               yields  g := x + 5
Divide(a,g,'q') mod 11;             yields  true
q;                                  yields  -2*x^2 - 3*x - 1
```

The order of precedence of all operators is described in section 3.2.13 below. However, any expression may be enclosed in parentheses yielding a new valid expression and this mechanism can be used to force a particular order of evaluation.

3.2.6. Neutral Operators

There is a facility for *user-defined* or *neutral* operators. A neutral operator symbol is formed by the ampersand character '&' followed by one or more characters which do not appear in the exclusion list below. The exclusion list consists of the ampersand character, all punctuation marks which are

not also programming-language operators, the comment character, and the white-space characters. Specifically, the characters excluded from &-names (following the &) are:

& | ( ) [ ] { } ; : ' ` # *newline blank.*

As with strings, the maximum length of an &-name is 495 characters.

Neutral operators may be used as unary prefix operators, or infix binary operators, or as function calls. They generate function calls, with the name of the function the same as the name of the neutral operator. (In the usual prettyprinting mode, these particular function calls are printed in binary operator format when there are exactly two operands, but the internal representation is an unevaluated function.) Maple imposes no semantics on the neutral operators. The user may define manipulations on expressions containing such operators via Maple's interface to user-defined procedures for various standard library functions, including simplify, diff, taylor, evalf, etc. (See chapter 7 for a more detailed discussion of operators.)

Examples:

a &* b;	yields	a &* b
&%1 a;	yields	&%1(a)
a * b &+ c;	yields	a*(b &+ c)
&–(a,b,c,d);	yields	&–(a,b,c,d)
a &xx b - &xx(a,b);	yields	0

3.2.7. Relations and Logical Operators

A new type of expression can be formed from ordinary algebraic expressions by using the *relational operators* <, <=, >, >=, =, <>. The semantics of these operators is dependent on whether they occur in an *algebraic* context or in a *boolean* context.

In an algebraic context, the relational operators are simply 'place holders' for forming equations or inequalities. Addition of equations/inequalities and multiplication of an equation/inequality by an algebraic expression are fully supported in Maple. In the case of adding or subtracting two equations, for example, the addition or subtraction is applied to each side of the equations yielding a new equation. In the case of multiplying an equation by an expression, the multiplication is distributed to each side of the equation. Similar operations may be performed with inequalities.

In a boolean context, a relation is evaluated to the value *true* or the value *false*. (The names *true* and *false* are defined to be constants in Maple.) In the case of the operators <, <=, >, >= the difference of the operands must evaluate to a numeric constant and this constant is compared with zero. In the case of either of the relations

```
op1 = op2
op1 <> op2
```

the operands may be arbitrary expressions (algebraic or non-algebraic). Note, however, that this equality test for expressions deals only with syntactic equality of the Maple representations of the expressions, which is not the same as mathematical equivalence. For example, the equation

```
x + y = y + x
```

will evaluate to *true* in a boolean context (because the Maple internal representations of these two expressions will be identical). However, the equation

```
x^2 - y^2 = (x - y) * (x + y)
```

will evaluate to *false* in a boolean context, even though the expressions are mathematically equivalent. In the latter example, applying the **expand** function results in an equation which will evaluate to *true*.

More generally, an expression can be formed using the *logical operators*

```
and
or
not
```

where the first two are binary operators and the third is a unary (prefix) operator. An expression containing one or more logical operators is automatically evaluated in a boolean context. (See section 3.2.13 regarding the order of evaluation.)

3.2.8. Ranges

Yet another type of expression is a *range* which is formed using the ellipsis operator:

```
[[expression]] .. [[expression]]
```

(the operator here can be specified as two *or more* consecutive periods). Any

valid Maple [[expression]]s may appear as the operands. The ellipsis operator simply acts as a 'place holder' in the same manner as when the relational operators are used in an algebraic context. To convert a range into an expression sequence, the $ operator may be applied (see section 3.2.3 above). For example:

```
1..5;     yields   1 .. 5
$1..5;    yields   1, 2, 3, 4, 5
```

Two common uses of ranges are for Maple's library functions **sum** and **int**. For example, in the function call

```
sum( i^2, i = 1 .. n )
```

the sum function interprets this to mean that the lower and upper limits of summation are 1 and n, respectively. Similarly, in the function call

```
int( exp(2*x), x = 3/4 .. 7/8 )
```

the integration function interprets this as a definite integration with lower and upper limits of integration 3/4 and 7/8, respectively. The range construct is also used by Maple's built-in function **op**, which extracts operands from an expression. For example, if

```
a := [ x, y, z, w ];
```

then

```
op(2, a)       yields   y
op(3, a)       yields   z
op(2 .. 4, a)  yields   y, z, w
```

The [[range]] construct can also be used in combination with the concatenation operator to form an [[expression sequence]], as follows. The construct

```
[[name]] . ( [[range]] )
```

is a generalization of name formation using the concatenation operator. Here, the left and right operands in [[range]] must be integers, and the result is an [[expression sequence]] consisting of the names formed by concatenating [[name]] to each integer in the specified range. If the right operand in [[range]] is less than the left operand then the result is the null expression sequence. For example,

```
p . (1 .. 5)
```

is equivalent to: p1, p2, p3, p4, p5 .

3.2.9. Selection Operation

The selection operation '[ ]' can be used to select components from an aggregate object. The syntax for the selection operation is

⟦name⟧ [⟦expression sequence⟧] .

The ⟦name⟧ must evaluate to one of the following:

name, table, array, list, set, expression sequence .

If ⟦name⟧ evaluates to an unassigned name, then ⟦name⟧[⟦expression sequence⟧] is an indexed name as described earlier (see section 3.2.2). If ⟦name⟧ evaluates to a table or an array, then the selection operation is an indexing operation. The use of arrays, tables, and indexing operations is discussed in chapter 5. If ⟦name⟧ evaluates to a list, set, or expression sequence, then ⟦expression sequence⟧ must evaluate to an integer, a range, or null. If ⟦expression sequence⟧ is an integer i, then the i^{th} operand of the aggregate object is returned. If ⟦expression sequence⟧ is a range, then an expression sequence is returned containing the operands of the aggregate object as specified by the range. If ⟦expression sequence⟧ is empty, then an expression sequence containing all of the operands of the aggregate object is returned. Thus there is the following explicit relationship between the selection operation and the **op** function (see section 9.1) in the case where ⟦name⟧ evaluates to a list or a set:

⟦name⟧ [⟦expression sequence⟧]

is equivalent to

op(⟦expression sequence⟧, ⟦name⟧) .

For example, if

```
a := [w,x,y,z];
```

then

```
a[3];      yields    y
a[1..3];   yields    w, x, y
a[ ];      yields    w, x, y, z
```

3.2.10. Unevaluated Expressions

An expression enclosed in a pair of single quotes is called an *unevaluated expression*. For example, the statements

```
a := 1; x := a + b;
```

cause the value 1+b to be assigned to the name x while the statements

```
a := 1; x := 'a + b';
```

cause the value a+b to be assigned to the name x. The latter effect can also be achieved (if b has no value) by the statements

```
a := 1; x := 'a' + b;
```

The effect of evaluating a quoted expression is to strip off (one level of) quotes, so in some cases it is useful to use nested levels of quotes. Note that there is a distinction between 'evaluation' and 'simplification' so that the statement

```
x := '2 + 3';
```

will cause the value 5 to be assigned to the name x even though the expression appearing here is quoted. The evaluator simply strips off the quotes but it is the 'simplifier' which transforms the expression 2 + 3 into the constant 5.

A special case of 'unevaluation' arises when a name which may have been assigned a value needs to be unassigned, so that in the future the name simply stands for itself. This is accomplished by assigning the quoted name to itself. For example, if the statement

```
x := 'x';
```

is executed, then even if x had previously been assigned a value it will now stand for itself in the same manner as if it had never been assigned a value. (See also the library function **evaln** in section 9.1.)

3.2.11. Procedures and Functions

Another valid expression in Maple is a *procedure definition* which takes the form

proc ([[nameseq]]) local [[nameseq]]; options [[nameseq]]; [[statseq]] end

where the 'local [[nameseq]];' part and the 'options [[nameseq]];' part may be

omitted. Here, ⟦nameseq⟧ stands for a sequence of ⟦name⟧s (specifically, they must be sequences of ⟦string⟧s) and in the first instance above, the ⟦nameseq⟧ may be empty. This construct has some similarities with the concept of unevaluated expressions, but in this case it is more generally a ⟦statseq⟧ (i.e., a sequence of statements) which is unevaluated. Note that the keywords 'proc' and 'end' serve a purpose similar to the single quotes in unevaluated expressions (except that evaluation of this expression does not cause these keywords to be stripped off). An example of a procedure definition is

```
max := proc ( a, b ) if a>b then a else b fi end
```

which is syntactically an assignment statement where the ⟦expression⟧ on the right hand side is a procedure definition.

A procedure is invoked by a *procedure invocation* which normally has the syntax:

⟦name⟧ (⟦expression sequence⟧) .

For example, if max is defined as above then the expression max(1, 2) is a procedure invocation in which the 'actual parameters' 1 and 2 are substituted for the 'formal parameters' a and b, respectively, and then the 'procedure body' is executed, yielding the value 2 in this case. (For a more general discussion of procedures, see chapter 6.)

The above syntax for a procedure invocation may also be used in cases where the ⟦name⟧ has not been assigned, in which case the result is an *unevaluated function*, which is another instance of a Maple expression. Examples of unevaluated functions are: sin(x), exp(x^2), and f(x,y).

More generally, a procedure invocation may take the form:

⟦expr⟧ (⟦expression sequence⟧)

where ⟦expr⟧ may be any of the following types of expressions:

name
procedure invocation
procedure definition
functional operator
integer
float
parenthesized algebraic expression.

For example, the library function **randpoly** is not readlib-defined (see chapter

9) so it must be explicitly loaded using **readlib(randpoly)**. Rather than the pair of statements

```
readlib(randpoly): p := randpoly(x);
```

it is possible to use the single statement

```
p := readlib(randpoly)(x);
```

to achieve the same action. An example of using a procedure definition directly in a procedure invocation is:

```
a := proc(x) x^2 end (5);
```

which assigns the value 25 to 'a'. The other cases listed above are discussed in the context of Maple's algebra of functional operators (see section 3.2.12 below and chapter 7).

3.2.12. Functional Operators

Maple supports a notation for the representation and manipulation of functional operators. Any of the following four forms of *angle-bracket notation* may be used to form a functional operator from an expression ⟦expr⟧:

```
< ⟦expr⟧ | ⟦nameseq1⟧ | ⟦nameseq2⟧ >
< ⟦expr⟧ | ⟦nameseq1⟧ >
< ⟦expr⟧ | | ⟦nameseq2⟧ >
< ⟦expr⟧ > .
```

The semantics of the angle-bracket notation can best be thought of as a process of *procedurizing* the given expression ⟦expr⟧. The internal data structure formed for the first case above is the following Maple procedure definition:

```
proc ( ⟦nameseq1⟧ ) local ⟦nameseq2⟧; options operator; ⟦expr⟧ end .
```

In the second case above, there are no local variables specified so the procedure definition is:

```
proc ( ⟦nameseq1⟧ ) options operator; ⟦expr⟧ end
```

and in the third case, it is:

```
proc ( ) local ⟦nameseq2⟧; options operator; ⟦expr⟧ end .
```

In the fourth case, no local variables are assumed, and the sequence of formal parameters is taken to be the name indeterminates appearing in ⟦expr⟧ as would be determined by invoking: **op(indets(⟦expr⟧,name))**. (See

chapter 9 for descriptions of **op** and **indets**.) Since the order of the formal parameters will be arbitrarily determined by Maple, this case is most useful when only one name indeterminate appears in ⟦expr⟧. With ⟦nameseq1⟧ determined in this manner, the procedure definition for the fourth case is identical with that of the second case.

Maple performs some automatic simplifications on functional operators (more precisely, on procedures with option operator). For example,

```
F := <sin(x) | x>;    yields   F := sin
```

where the functional operator has been simplified to the function name sin. The reason is that the action of applying the operator F to an argument will be precisely the action of applying the function name sin to the argument. Further examples are:

```
A := <1>;       yields  A := 1
B := <3.14>;    yields  B := 3.14
```

which are examples of *constant operators*. The result of applying a constant to some arguments in a functional notation is the constant itself:

```
<1>(x);          yields  1
1(x);            yields  1
<3.14>(x,y);     yields  3.14
3.14(x,y);       yields  3.14 .
```

Note that the *identity operator* is expressed as <x> for any choice of parameter name x. Note also that parenthesized algebraic expressions, when applied to some arguments in a functional notation, are interpreted in an "operator algebra".

Examples:

```
<x>(t);                          yields   t
(a+b)(t);                        yields   a(t) + b(t)
(a+1)(t);                        yields   a(t) + 1
( <ln(x)+1>@@2 )(t);             yields   ln(ln(t) + 1) + 1
( <sin(x)>@<arcsin(x)> )(t);     yields   t
```

(See chapter 7 for a more detailed discussion of functional operators.)

3.2.13. Precedence of Programming-Language Operators

The order of precedence of all unary and binary operators is listed in the following table, from highest to lowest binding strengths. In parentheses it is stated whether the operators are left associative, right associative, or non-associative. Note, however, that any expression may be enclosed in parentheses yielding a new valid expression and this mechanism can be used to force a particular order of evaluation.

.	(left associative)
&-operators	(left associative)
!	(left associative)
^, **, @@	(non-associative)
*, /, @, intersect	(left associative)
+, −, union, minus	(left associative)
mod	(non-associative)
..	(non-associative)
<, <=, >, >=, =, <>	(non-associative)
$	(non-associative)
not	(right associative)
and	(left associative)
or	(left associative)
,	(left associative)
:=	(non-associative)

Thus the concatenation or decimal point operator '.' has the highest binding strength and the assignment operator ':=' has the lowest binding strength. Note that the exponentiation operators '^', '**', and '@@' are defined to be non-associative and therefore a^b^c is syntactically invalid in Maple. (The user must use parentheses to state his intentions.)

The evaluation of expressions involving the logical operators proceeds in an intelligent manner which exploits more than the simple associativity and precedence of these operators. Namely, the left operand of the operators 'and' and 'or' is always evaluated first and the evaluation of the right operand is avoided if the truth value of the expression can be deduced from the value of the left operand alone. For example, the construct

```
if d <> 0 and f(d)/d > 1 then . . . fi
```

will not cause a division by zero, because if d=0 then the left operand of 'and' becomes false and the right operand of 'and' will not be evaluated.

3.3. Sample Maple Session

This section presents a sample interactive session using the Maple system. Typically, Maple can be initiated by the command `maple` but this depends on local installation details on the particular host computer system. In the following presentation of the Maple session, lines beginning at the left margin immediately following the prompt symbol '> ' are user input lines and centered lines are system responses. Each user input line must be terminated by [[return]] or [[newline]].

The sample computations shown in the following Maple session are algebraically trivial and are designed simply to show some of the features of the language. The reader should consult the document *First Leaves: A Tutorial Introduction to Maple*[Cha88] for an extensive introduction to Maple using sample problems.

```
    |\^/|
._|\|   |/|_. Math Faculty, Univ of Waterloo
 \  MAPLE  /  Version 4.2 --- Dec 1987
 <____ ____>  For on-line help, type  help();
      |
# Integers, rational numbers, and floating-point numbers.
> 254 + 5280*99999;
                          527994974

> 3!;
                              6

> 3!!;
                             720

> 1 + 1/4 + 1/16 + 1/64 + 1/256;
                             341
                            -----
                             256

> evalf(");
                         1.332031250
```

```
> a := (5^40 + 3^50) / 2^90;
                    45478324578584871158695804 37
              a := ------------------------------
                    61897001964269013744956211 2

> evalf(a);
                          7.347419606

> Digits := 30;
                        Digits := 30

> evalf(a);
                7.34741960601547810893226043509

> evalf(a, 50);
   7.34741960601547810893226043508788811613237847869 02

> b := 2^90;
              b := 1237940039285380274899124224

> a*b;
                909566491571697423173916087 4
```

Names, including the concatenation operator.

```
> g := 52;  G := 4;
                           g := 52

                           G := 4

> g*G;
                             208

> for i to 5 do p.i := i^2 od:
> p.(1..5);
                       1, 4, 9, 16, 25

> x := 333;  x := 'x';
                          x := 333

                           x := x
```

```
# Sets and lists.
> set1 := {x, 2*y+1/3};  set2 := {z-4, x};
                  set1 := {2 y + 1/3, x}

                    set2 := {z - 4, x}

> set1 union set2;
                  {2 y + 1/3, z - 4, x}

> set1 intersect set2;
                              {x}

> set3 := set1 minus set2;
                    set3 := {2 y + 1/3}

> set2 intersect set3;
                              {}

> list1 := [x, 2*y+1/3];  list2 := [z-4, x];
                  list1 := [x, 2 y + 1/3]

                    list2 := [z - 4, x]

> new_list := [op(list1), op(list2)];
          new_list := [x, 2 y + 1/3, z - 4, x]

# Polynomials and rational functions.
> p := x^2 - x - 2;
                           2
                     p := x  - x - 2

> q := (x+1)^2;
                                 2
                      q := (x + 1)

> r := "" * "";
                      2                   2
               r := (x  - x - 2) (x + 1)

> expand(r);
                 4    3      2
                x  + x  - 3 x  - 5 x - 2
```

```
> s := p/q;
```

$$s := \frac{x^2 - x - 2}{(x+1)^2}$$

```
> normal(s);
```

$$\frac{x - 2}{x + 1}$$

```
> r*s;
```

$$(x^2 - x - 2)^2$$

Equations.

```
> eqn1 := 3*u + 5*v = 13;
```

$$eqn1 := 3u + 5v = 13$$

```
> eqn2 := 4*u - 7*v = 30;
```

$$eqn2 := 4u - 7v = 30$$

```
> 3*eqn2 - 4*eqn1;
```

$$-41v = 38$$

```
> v := 38/(-41);
```

$$v := -\frac{38}{41}$$

```
> eqn1;
```

$$3u - \frac{190}{41} = 13$$

```
> u := (13 + 190/41)/3;
```

$$u := \frac{241}{41}$$

```
> eqn1;  eqn2;
                              13 = 13

                              30 = 30

> solve( {5*x + 10*y = 97,  x - y = 12},  {x,y} );
                            217        37
                       {x = -----, y = ----}
                            15         15

# Unevaluated expressions and procedures.
> a; b;
                  45478324578584871158695804 37
                 ----------------------------------
                  6189700196426901374495621 12

                  1237940039285380274899124224

> f := 'b * (a+5)';
                          f := b (a + 5)

> f;
                  1528536511214387560623478199 4

> max := proc (a,b) if a>b then a else b fi end;
max := proc (a, b) if b < a then a else b fi end

> max(a,b);
                  1237940039285380274899124224

> max(25/7, 525/149);
                               25/7
```

```
# Integers can be arbitrarily long. For example:
> 720!;
```

```
2601218943565795100204903227081043611191521875016945785\
7275418378508356311569473822406785779581304570826199205\
7589224725953664156516205201587379198458774083252910524\
4690388811884123764341191951045505346658616243271940197\
1139098455367272785370993456298555867193697740700037004\
3078375899742067678401696720784628062922903210716166986\
7260548988445514257193985499448939594496064045132362140\
2659861930732493697704776060676806701764916694030348199\
6188145562519559256691883082551494294759653727484562462\
8824234526597789737740896466553992435928786212515967483\
2209760295056966999272846705637471375330192483135870761\
2541268341586012944756601145542074958995256354306828863\
4631084965650682771552996256790845235702552186222358130\
0167008345234432368219357931847019565107297818043541738\
9056072742804858399591972902172661229129842051606757903\
6232337699453964191475175567557695392233803056825308599\
9774416757843528159134613403946049012695420288383471013\
6373382448450666009334848444071193129253769465735433737\
5724772230181534032647177531984537341478674327048457983\
7866187032574059389242157096959946305575210632032634932\
0922073832092335630992326750440170176057202601082928804\
2335606643089888710297380797578013056049576342838683057\
1906622052911748225105366977566030295740433879834715185\
5260280533386635713910104633641976909739743228599421983\
7046979109956303389604675889865795711176566670039156748\
1531159439800436253993997312030664906013253113047190288\
9849185620376666916446879112524919375442584589500031156\
1682974304641142538074897281723375955380661719801404677\
9356147936352662656833395097600000000000000000000000000\
0000000000000000000000000000000000000000000000000000000\
0000000000000000000000000000000000000000000000000000000\
0000000000000000000000000000000000000000
```

```
# Compute the factorization of the above integer.
> ifactor(");
```

```
   716     356     178     118      70      59      44      38
(2)     (3)     (5)     (7)     (11)    (13)    (17)    (19)

       32      24      23      19      17      16      15
   (23)    (29)    (31)    (37)    (41)    (43)    (47)
```

```
    13      12      11      10      10      9       9       8
(53)    (59)    (61)    (67)    (71)    (73)    (79)    (83)

    8       7       7       6       6       6       6
(89)    (97)    (101)   (103)   (107)   (109)   (113)

    5       5       5       5       4       4       4
(127)   (131)   (137)   (139)   (149)   (151)   (157)

    4       4       4       4       3       3       3
(163)   (167)   (173)   (179)   (181)   (191)   (193)

    3       3       3       3       3       3       3
(197)   (199)   (211)   (223)   (227)   (229)   (233)

    3       2       2       2       2       2       2
(239)   (241)   (251)   (257)   (263)   (269)   (271)

    2       2       2       2       2       2       2
(277)   (281)   (283)   (293)   (307)   (311)   (313)

    2       2       2       2       2       2       2
(317)   (331)   (337)   (347)   (349)   (353)   (359)

(367)  (373)  (379)  (383)  (389)  (397)  (401)  (409)

(419)  (421)  (431)  (433)  (439)  (443)  (449)  (457)

(461)  (463)  (467)  (479)  (487)  (491)  (499)  (503)

(509)  (521)  (523)  (541)  (547)  (557)  (563)  (569)

(571)  (577)  (587)  (593)  (599)  (601)  (607)  (613)

(617)  (619)  (631)  (641)  (643)  (647)  (653)  (659)

(661)  (673)  (677)  (683)  (691)  (701)  (709)  (719)
```

4. DATA TYPES AND FORMAL SYNTAX

4.1. Data Types

Every expression in Maple is represented internally by an expression tree where each node is a particular data type. While some data types are strictly for internal use, most of the data types corresponding to expressions are accessible to the user and can be tested for via the **type** function. The user can examine the components of such a data type by using the **op** function and the corresponding **nops** function can be used to query the "number of operands" of an expression. In this section we discuss the data types which are accessible to the user. For a more detailed description of the internal data types and their representation, see chapter 8.

The syntax and functionality of the **type**, **op**, and **nops** functions are described in section 9.1. Since the following descriptions make frequent references to the **op** function, the following details should be noted. There are three forms used:

op(i, expr); op(i..j, expr); op(expr);

where i and j must be nonnegative integers. The first form is used to extract the i^{th} operand from expr. Typically i is greater than or equal to one, but for some data types a $zero^{th}$ operand is defined as noted in the following subsections. The second form is used to extract the i^{th} through j^{th} operands from expr, yielding an expression sequence containing $j-i+1$ subexpressions, and the result is the null expression sequence if $j < i$. The third form is equivalent to op(1..n, expr) where n is the number of operands appearing in expr (ignoring the $zero^{th}$ operand if it is defined).

4.1.1. Integer

An expression is of type 'integer' if it is an (optionally signed) sequence of one or more digits of arbitrary length. (The length limit is system dependent but generally much larger than users will encounter – typically greater than 250,000 decimal digits.) The **op** function considers this data type to have only one operand, so if n is an integer then the value of op(n), and also the value of op(1, n), is the integer n.

4.1.2. Fraction

An expression of type 'fraction' is represented by a pair of integers (numerator and denominator) with all common factors removed and with a positive denominator. Like integers, fractions are of arbitrary length. The

op function considers this data type to have two operands, where the first operand is the numerator and the second operand is the denominator.

The **type** function also understands the composite type 'rational', which is defined to be either an 'integer' or a 'fraction'.

4.1.3. Floating-Point Number

A floating-point number (called type 'float') is represented externally as a sequence of digits with a decimal point, possibly multiplied by a power of 10. (E.g., 1.5, 15000., .15, .1500000000*10^15.) Floating-point numbers are represented internally by a pair of integers (the mantissa and the exponent), which represent the number

$$\text{mantissa} \times 10^{\text{exponent}}.$$

The exponent in this representation is restricted to be a *word-size* integer on the host computer. (This size limit is system-dependent but typically it is 9 or 10 digits in length.) Thus the **op** function considers this data type to have two operands. For example, op(150.1) yields the expression sequence 1501, −1.

For arithmetic operations and invocations of standard functions, if one of the operands is a floating-point number then floating-point evaluation will take place automatically. More generally, the **evalf** function may be used to force evaluation to a floating-point number. The number of digits carried in the 'mantissa' when evaluating floating-point numbers is determined by the value of the global name 'Digits' which has 10 as its initial value.

An alternative input format for floating-point numbers in Maple is to use the notation Float(mantissa, exponent) which directly corresponds to Maple's internal data structure representation of floating-point numbers (where both arguments to the Float function must be integers). When Maple's line-printing mode is used (via the **lprint** function, or in normal display when pretty-printing is turned off by assigning zero to the global name *prettyprint*), the Float(mantissa, exponent) notation is used for floating-point numbers with very small or very large magnitudes. For example,

```
prettyprint := 0: evalf(exp(-10)); evalf(exp(-20));
```

yields

```
Float(4539992976,-14)
Float(2061153622,-18)
```

in contrast with the normal display of these numbers which would be seen in pretty-printed format as:

.00004539992976

$.2061153622*10^{-8}$

The **type** function also understands the composite type 'numeric', which is defined to be an 'integer', a 'fraction', or a 'float'.

4.1.4. String

An expression is of type 'string' if it is either a [[simple string]] or a [[quoted string]]. Specifically, a [[simple string]] is a sequence of one or more letters, digits, and underscores beginning with either a letter or an underscore; a [[quoted string]] is any sequence of characters in the supported character set formed by enclosing the characters within a pair of back quotes. The **type** function understands the composite type 'name', which is defined to be a 'string' or an 'indexed name'.

The string data type has only one operand which is the value assigned to it (in its role as a name) or else its own name if no other value has been assigned. For example, if x has not been assigned a value then x is of type 'string' and op(x) has the value x. Another example of a Maple string is `This is a string`. The maximum length of a string is 495 characters.

Note that the construct op('x') serves as a "one-level evaluation" operation in Maple, for any string x, since the argument will evaluate to the string x (not its value) and the "operand" of a string is defined to be the value assigned to it. In other words, this construct yields the value "pointed at" by the string (name) x without further evaluation of that value. For example,

```
x := y:
y := 2/3:
x;          yields   2/3
op('x');    yields   y
op(x);      yields   2, 3
```

Normal evaluation of x (i.e., full recursive evaluation) yields the value 2/3, but the **op** function with argument 'x' (unevaluated x) yields the value immediately assigned to x. In the last statement above, the argument to the **op** function is the value of x, namely 2/3, which has two operands.

4.1.5. Indexed Name

An indexed name, called type 'indexed', is of the form

[[name]] [[[expression sequence]]]

where [[name]] is an unassigned name. The [[name]] could be a string, or it could be an indexed name itself so there can be a succession of subscripts as in:

A[1,2,3][x,y][2,1] .

The **op** function applied to this data type will yield the expression sequence of indices appearing within the rightmost pair of brackets, and the 0^{th} operand is the corresponding [[name]]. For example, if expr evaluates to the above indexed name then

```
op(0, expr);        yields    A[1,2,3][x,y]
op(1, expr);        yields    2
op(2, expr);        yields    1
```

As mentioned in the preceding subsection, the **type** function understands the composite type 'name', which is defined to be a 'string' or an 'indexed name'.

4.1.6. Addition, Multiplication, and Exponentiation

An expression can be composed using the algebraic operators +, −, *, /, ^ (or **). Such an expression is of type `+`, type `*`, or type `^` (or type `**`). Thus the expression a−b is of type `+` and op(a−b) yields the expression sequence a, −b. Similarly the expression a/b is of type `*` and op(a/b) yields the expression sequence a, 1/b. The expression 1/b is an example of an expression of type `^` since it is represented internally as b^(−1). The representation used for these algebraic expressions is often referred to as sum-of-products form.

An expression of type `^` always has exactly two operands, the base expression and the exponent expression. An expression of either type `+` or type `*` can have two or more operands. For example:

```
op( x+y+z+w );          yields    x, y, z, w
op( 2*x^2*y );          yields    2, x^2, y
op( (x+y)^z );          yields    x + y, z
```

4.1.7. Series

The 'series' data type in Maple is a special data type which represents an expression as a (truncated) power series with respect to a specified indeterminate, expanded about a particular point. This data type is created by a call to the **taylor** function. For this data type, the $zero^{th}$ operand is

x–a where 'x' denotes the specified indeterminate and 'a' denotes the particular point of expansion. The first, third, . . . operands are the coefficients (general expressions) and the second, fourth, . . . operands are the corresponding integer exponents. The exponents are restricted to be *word-size* integers on the host computer (typically, the length limit is 9 or 10 digits) and they are ordered from least to greatest. Usually, the final pair of operands in this data type are the special 'order' symbol O(1) and the integer n which indicates the order of truncation. (Note: The print routine displays the final pair of operands using the notation $O(x^n)$ rather than more directly as $O(1)\ x^n$, where x represents the zeroth operand.) However, if the series is known to be exact then there will be no 'order' term in the series. An example of this occurs when the 'taylor' function is applied to a polynomial whose degree is less than the truncation degree for the series. A very special case is the zero series, which is immediately simplified to the integer zero in Maple.

The power series represented by this data structure are generalized power series which includes Laurent series with finite principal part but, more generally, the series coefficients are allowed to depend on x under the following restriction. Formally, if the coefficients c_i are dependent on x then for any $\epsilon > 0$

$$k_1\,(x-a)^{\epsilon} \;<\; |\,c_i\,| \;<\; k_2\,(x-a)^{-\epsilon} \quad \text{for all } 0 < x-a < r$$

for some positive constants k_1, k_2, and r. In other words, the coefficients may depend on x but their growth must be less than polynomial in x. O(1) represents such a coefficient, rather than an arbitrary constant. An example of a non-standard generalized power series is:

$$x^x = 1 + \ln(x)x + \frac{\ln(x)^2x^2}{2} + \frac{\ln(x)^3x^3}{6} + \frac{\ln(x)^4x^4}{24} + O(x^5)\ .$$

The **type** function also understands the composite type 'algebraic', which corresponds to all of the data types discussed in sections 4.1.1 through 4.1.7, plus type 'function' (section 4.1.15), type `.` (section 4.1.16) and type 'uneval' (section 4.1.17).

4.1.8. Relation

An expression of type 'equation' (also called type `=`) has two operands, the left-hand-side expression and the right-hand-side expression. An equation is represented externally using the binary operator '='.

There are three internal data types for inequalities, corresponding to the operators '<>', '<', and '<='. Inequalities involving the operators '>' and '>=' are converted to the latter two cases for purposes of

representation. Correspondingly, only three names are known to the **type** function for inequalities: `<>`, `<`, `<=` . Like an equation an inequality has two operands, the left-hand-side expression and the right-hand-side expression.

The **type** function also understands the composite type 'relation', which is defined to be one of: `=`, `<>`, `<`, `<=` .

4.1.9. Boolean Expression

The simplest cases of boolean expressions are the names true and false. These two names are defined to be constants in Maple. Equations and inequalities (formed using the relational operators =, <>, <, <=, >, >=) are also treated as boolean expressions if they appear in a 'boolean context'. More complicated boolean expressions can be built out of these simple expressions with the logical operators **and**, **or**, and **not**. The built-in function **evalb** can be called with a boolean expression as argument in order to cause the expression to be evaluated as a boolean. For example, the equation a=b is an algebraic equation if it appears alone but evalb(a=b) will evaluate this equation as a boolean. However, an equation or inequality will be recognized as being in a boolean context if it appears in the 'while part' of a repetition statement, in the 'if part' of a selection statement, or if it is an operand in an expression formed using one or more of the logical operators: and, or, not.

In addition to the type names for relations, the following type names are known to the **type** function: `and`, `or`, `not` . Each of the first two types of expressions has exactly two operands which can be selected by the **op** function, and the third type has exactly one operand. For example:

```
op( a and b and c );      yields    a and b, c
op( a or b and c );       yields    a, b and c
op( not(a and b) );       yields    a and b
```

The **type** function also understands the composite type 'boolean', which is defined to be one of the names true or false, or one of the types: `and`, `or`, `not`, `=`, `<>`, `<`, `<=` .

4.1.10. Range

An expression of type 'range' (also called type `..`) has two operands, the left-hand-side expression and the right-hand-side expression. A range is represented externally using the binary operator '..' which simply acts as a place-holder in the same manner as the relational operators. For example, the range 1..3 is not equivalent to the expression sequence 1, 2, 3. In

certain contexts it is understood to represent this sequence, such as in the construct p.(1..3) or in the construct sum(i, i = 1..3). However, if the actual expression sequence is desired, the sequence operator $ (see section 3.2.3) can be used in the form: $1..3.

4.1.11. Expression Sequence

There is a data type, called type 'exprseq', for an ⟦expression sequence⟧, which is a sequence of ⟦expression⟧s separated by commas. The operands appearing in an expression sequence cannot be directly extracted via the **op** function due to the nature of an expression sequence. For example,

```
s := x, y, z;
op(2, s);
```

yields

```
Error, wrong number (or type) of parameters in function op;
```

This is because the arguments to the **op** function are specified by the expression sequence 2, x, y, z which is too many arguments for the **op** function. If it is desired to extract the second operand from the expression sequence s, this can be accomplished via **op** if the expression sequence is placed into a list:

```
op(2, [s]); yields   y
```

Alternatively, the *selection operation* discussed in section 3.2.9 may be used without forming a list:

```
s[2]; yields   y
```

When the **op** function is used to extract parts of an expression, the result is often an expression sequence. For example,

```
a := [x,y,z,w]; op(a);
```

yields the expression sequence x, y, z, w . An important special case of an expression sequence is the null expression sequence and there is a global name in Maple, NULL, whose value is the null expression sequence. The value of the global name NULL is equivalent to the value of the operation op([ ]).

4.1.12. Set and List

Two more data types are the 'set' and the 'list'. Each of these types consists of a sequence of expressions and if expr is an object of either of these two types then op(expr) yields the expression sequence. The external representation of a set uses braces '{', '}' to surround the expression sequence and the external representation of a list uses brackets '[', ']' to surround the expression sequence. The empty set is represented by { } and the empty list is represented by [ ].

4.1.13. Table and Array

The table data type in Maple is created either explicitly via the **table** function call or implicitly by assigning to an indexed name. For example, the statement

```
b := table( [(1)=x] );
```

creates a table object with one component and assigns the table object to the name b, while the following statement has precisely the same effect if b has no previous value:

```
b[1] := x;
```

The array data type in Maple is a specialization of the table data type. Specifically, a Maple array is a table with specified dimensions and each dimension is an integer range. An array is always created explicitly via the **array** function call. An array object consists of three parts: an indexing function, an index bound, and a collection of components. An array object has only two operands that can be selected by the **op** function: the first operand is the indexing function and the second operand is the index bound. Either of these operands may be NULL. The third part of the array object, the collection of components, is not accessible via the **op** function but is only accessible via the indexing (selection) operation.

For example, the statement

```
A := array( symmetric, 1..2,1..2, [[x,y],[y,z]] );
```

assigns to the name A a symmetric, 2-by-2 array whose first row contains the values x, y and whose second row contains the values y, z. The following statement would have the same effect:

```
A := array( symmetric, 1..2,1..2, [(1,1)=x, (1,2)=y, (2,2)=z] );
```

In either case,

op(1, op(A));	yields	symmetric
op(2, op(A));	yields	1 .. 2, 1 .. 2
A[2,1];	yields	y

Note the use of the construct op(A) here. Normally in Maple a name evaluates to its value, but if the value is a table (array) or a procedure then the result of evaluation is the name itself. Hence in the above example, we must specify op(A) in order to extract operands from the actual array structure rather than the name A.

For a non-array table structure, there is only one operand accessible to the **op** function; namely, the indexing function (which may be NULL).

Maple uses a sparse representation for tables and arrays, applying a hashing technique for the internal representation. Thus there is no memory space reserved for components that have not been assigned. (See chapter 5 for a detailed discussion of arrays and tables.)

4.1.14. Procedure Definition

A procedure definition in Maple is a valid expression and its type is called 'procedure'. The external representation of a procedure definition is

proc ([[nameseq]]) local [[nameseq]]; options [[nameseq]]; [[statseq]] end

The internal data structure represents each [[nameseq]] in the order shown above, followed by a (possibly null) table of remembered values, followed by the statement sequence [[statseq]]. Since [[statseq]] is not a valid expression in Maple, this part of the data structure is not retrievable by the **op** function. When the **op** function is applied to this data structure, there are four operands of the structure that can be accessed: the first operand is the [[nameseq]] of formal parameters, the second operand is the [[nameseq]] of local variables, the third operand is the [[nameseq]] of option names, and the fourth operand is the remember table for the procedure. Any of these operands may be NULL.

For example, in the procedure expression

proc ( ) end

all four operands are NULL. As another example consider

```
f := proc (a,b)
  local c;
  option remember;
  c := a/b;
  if type(c, integer) then c else RETURN('procname(args)') fi
end;
```

For the above procedure

```
op(1, op(f));   yields   a,b
op(2, op(f));   yields   c
op(3, op(f));   yields   remember
```

Furthermore, if the invocations f(2,4) and f(4,2) have been executed, then

```
op(4, op(f));      yields      table([(2, 4)=f(2,4),(4, 2)=2]) .
```

Note the use of the construct op(f) here. Normally in Maple a name evaluates to its value, but if the value is a procedure or a table then the result of evaluation is the name itself. Hence in the above example, we must specify op(f) in order to extract operands from the actual procedure definition rather than the name f. (See chapter 6 for a detailed discussion of procedures.)

4.1.15. Unevaluated Function Invocation

One form of function invocation is

[[name]] ([[expression sequence]])

and if [[name]] is undefined then the result is an unevaluated function invocation, called type 'function'. Typical examples of the type 'function' are sin(x), exp(x^2), and g(a,b) where none of sin, exp, and g has been defined. When the **op** function is applied to this data type, operand 0 is defined to be the name of the function and the remaining operands are the elements of the [[expression sequence]]. For example,

```
op(0, g(a,b));   yields   g
op(1, g(a,b));   yields   a
op(2, g(a,b));   yields   b
op(g(a,b));      yields   a,b
```

The special type `!` is also known to the **type** function. This corresponds to the unevaluated function named 'factorial' but since the special operator '!' is usually used for factorials, the special type name has been provided. For example, if n is undefined then

```
type( n!, `!` )          yields   true
type( n!, function );    yields   true
op( 0, n! );             yields   factorial
factorial( n );          yields   n!
```

4.1.16. Unevaluated Concatenation

An expression which consists of an unevaluated concatenation is said to be of type `.` . Normally, the concatenation operator is evaluated to form a name but an example of an expression of type `.` would be a.(i^2), where i has no value. This data type has two operands which may be extracted via the **op** function; namely, the left-hand-side expression and the right-hand-side expression.

4.1.17. Unevaluated Expression

There is a data structure for unevaluated expressions, called type 'uneval', corresponding to expressions which have been quoted (using single quotes) to prevent evaluation. For example, the expression resulting from the evaluation of

```
"x + y"
```

is the expression 'x + y' which is of type 'uneval'. This data type has just one operand which may be extracted by the **op** function; namely, the expression inside the (outermost) pair of single quotes.

4.2. Formal Syntax

This section presents the BNF grammar which describes the syntax accepted by Maple. In the following grammar, where a sequence of symbols is enclosed in a pair of "§" symbols it indicates that this portion of the statement is optional. Where *empty* occurs in the grammar, no symbol is required. A Maple *session* consists of a ⟦statseq⟧, which is a sequence of statements separated by semicolons or colons. At the top (interactive) level of the Maple system, the colon serves to suppress the display of the result of the preceding statement. Within Maple procedures, the colon and the semicolon are interchangeable.

⟦statseq⟧ ::= ⟦statseq⟧ ; ⟦stat⟧ | ⟦statseq⟧ : ⟦stat⟧
| ⟦stat⟧

⟦stat⟧ ::= ⟦name⟧ := ⟦expr⟧
| ⟦expr⟧
| **read** ⟦expr⟧
| **save** ⟦expr⟧
| **if** ⟦expr⟧ **then** ⟦statseq⟧ ⟦elsepart⟧
| § **for** ⟦name⟧ § § **from** ⟦expr⟧ § § **by** ⟦expr⟧§
§ **to** ⟦expr⟧ § § **while** ⟦expr⟧ §
do ⟦statseq⟧ **od**
| § **for** ⟦name⟧ § § **in** ⟦expr⟧ § § **while** ⟦expr⟧ §
do ⟦statseq⟧ **od**
| **quit***
| *empty*

⟦expr⟧ ::= ⟦expr⟧ , ⟦expr⟧ *expression sequence*
| ⟦expr⟧ **or** ⟦expr⟧ *boolean expressions*
| ⟦expr⟧ **and** ⟦expr⟧ | **not** ⟦expr⟧
sequence generators
| ⟦expr⟧ **$** ⟦expr⟧ | **$** ⟦expr⟧
relations
| ⟦expr⟧ < ⟦expr⟧ | ⟦expr⟧ <= ⟦expr⟧
| ⟦expr⟧ > ⟦expr⟧ | ⟦expr⟧ >= ⟦expr⟧
| ⟦expr⟧ <> ⟦expr⟧ | ⟦expr⟧ = ⟦expr⟧
| ⟦expr⟧ ..† ⟦expr⟧ *range*
| ⟦expr⟧ **mod** ⟦expr⟧ *modulo operation*

---

* **done** or **stop** can be used as synonyms for **quit** . In many implementations, pressing the break/interrupt key (or some similar system-dependent sequence) twice in rapid succession will also cause Maple to exit. See the appendices for system-specific details.

† Actually, two *or more* consecutive periods are permitted.

+ ⟦expr⟧		*algebraic and set operations*
− ⟦expr⟧		
⟦expr⟧ + ⟦expr⟧	⟦expr⟧ − ⟦expr⟧	
⟦expr⟧ **union** ⟦expr⟧	⟦expr⟧ **minus** ⟦expr⟧	
⟦expr⟧ * ⟦expr⟧	⟦expr⟧ / ⟦expr⟧	
⟦expr⟧ @ ⟦expr⟧	⟦expr⟧ **intersect** ⟦expr⟧	
⟦expr⟧ ^ ⟦expr⟧	⟦expr⟧ ** ⟦expr⟧	
⟦expr⟧ @@ ⟦expr⟧		
⟦expr⟧ !		*factorial*
⟦neutralop⟧ ⟦expr⟧		*neutral operators*
⟦expr⟧ ⟦neutralop⟧ ⟦expr⟧		
proc (§ ⟦nameseq⟧ §) § **local** ⟦nameseq⟧ ; § § **options** ⟦nameseq⟧ ; §‡ ⟦statseq⟧ **end**		*procedure definition*
⟦name⟧		*variable name*
(⟦expr⟧)		*parenthesized expression*
' ⟦expr⟧ '		*unevaluated expression*
< ⟦expr⟧ >		*functional operators*
< ⟦expr⟧ | ⟦nameseq⟧ >		
< ⟦expr⟧ | § ⟦nameseq⟧ § | ⟦nameseq⟧ >		
⟦expr⟧ (⟦exprseq⟧)		*function call*
⟦natural⟧ . ⟦natural⟧		*floating-point numbers*
⟦natural⟧ .		
. ⟦natural⟧		
⟦natural⟧		*unsigned integer*

‡ **option** can be used as a synonym for **options** .

	{ [[exprseq]] }	*set*
	[[[exprseq]]]	*list*
	%	*previous expressions*
	%% | %%%	
[[elsepart]] ::=	**fi** | **else** [[statseq]] **fi**	
	elif [[expr]] **then** [[statseq]] [[elsepart]]	
[[exprseq]] ::=	[[expr]] | *empty*	
[[nameseq]] ::=	[[nameseq]] , [[string]] | [[string]]	
[[name]] ::=	[[string]] | [[name]] . [[natural]]	
	[[name]] . [[string]] | [[name]] . ([[expr]])	
	[[name]] [[[exprseq]]]	
	[[name]] ([[exprseq]])	
[[string]] ::=	[[simple string]] | [[quoted string]]	

[[simple string]] ::= *A sequence of one or more letters, digits, and underscores, beginning with either a letter or underscore*

[[quoted string]] ::= *Any sequence of characters in the supported character set enclosed within a pair of back quotes ` `*

[[neutralop]] ::= *The ampersand character & followed by one or more characters excluding:*

& | ( ) [ ] { } ; : ' ` # *newline blank*

[[natural]] ::= *A sequence of one or more of:*

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

5. ARRAYS AND TABLES

5.1. Overview

One of the data types in Maple is the *table* structure. The *array* structure in Maple is a specialization of the table data structure in which the indices must be integer expression sequences lying within user-specified bounds. Arrays are used similarly to those in other programming languages, while tables correspond roughly to the ones provided in Snobol or Icon.

As with other data types in Maple, tables are self-describing data objects, which may be created dynamically, passed as parameters, and so on. No declarations are needed; to make a name refer to a table, an assignment statement is used in which the right-hand side evaluates to a table object.

A table object consists of three parts:

- an indexing function
- an index bound (for arrays only)
- a collection of components

The indexing function allows a table to have a user-defined interface. A detailed discussion of indexing functions is given in section 5.5. If no special interface is to be used, the indexing function should have the value NULL. (This is the default.)

The only tables which have index bounds are arrays. The index bound is an expression sequence of integer ranges. The number of ranges is called the *dimension* of the array. Whenever a component of an array is referenced, the index is checked against the index bound. The *ith* range gives the bounds on the *ith* integer in the expression sequence used as the index.

The **op** function may be used to extract the operands of a table. The indexing function is available as the first operand of a table object. For arrays, the index bound is available as the second operand. The components are not available using the **op** function. The reason for this is that the collection of components is stored in an internal hash table and is not a user-level expression. (This is analogous to the statement sequence in a procedure not being available to the **op** function.)

Components of tables are accessed by using brackets ('[' and ']') for indexing. If the value of T is a table then components of T are referenced using the syntax T[[expression sequence]] . For example, executing the following statements

```
T[1,2] := a;  T[2,0] := b;  V := T[1,2]+T[2,0]
```

causes V to be assigned the value a+b.

Tables may be created either (i) explicitly, by calling one of the builtin functions **array** or **table**, or (ii) implicitly, by making an assignment to an indexed name of the form A[[expression sequence]] when A does not have a previous value. The creation of tables is described in section 5.2.

Expressions of type 'array' and of type 'table' are represented internally using the same data structure. The external representation is as a call to one of the functions **array** or **table** which would re-create the object. Specifically, the external representation for a table is:

table([[indexing function]], [[equation sequence]]) ,

where each equation in the [[equation sequence]] is of the form

(index) = component_value .

The equation sequence enclosed in brackets is a representation of an internal hash table. The equations will appear in an apparently arbitrary order which cannot be controlled by the user.

The external representation for an array is:

array([[indexing function]], [[range sequence]], [[values]])

where the [[values]] are presented (in Maple's pretty-printing mode) as a list in the case of a one-dimensional array, as a sequence of lists (row by row) in the case of a two-dimensional array, or as an ordered sequence of equations in the case of higher-dimensional arrays. In line-printing mode, the values of an array are printed as an unordered [[equation sequence]] as described above for general tables.

5.2. Creating Tables

5.2.1. Explicit Table Creation

The function **array** is used to create an array explicitly. To explicitly create a table which is not an array, the function **table** is used. These functions take a number of optional parameters which specify information about the table to be created.

The most common uses of these functions are illustrated by the following examples:

```
t := table( );
a := array(1..n);
b := array(1..n, 1..m);
```

Here, 't' has been assigned a new table object, 'a' has been assigned a one-dimensional array with n components, and 'b' has been assigned an n by m, two-dimensional array.

The **table** function, in general, takes two parameters: an indexing function and an initialization list. The function **array** takes an indexing function, an initialization list, and an index bound. The index bound is passed as a number of integer ranges appearing adjacently in the parameter sequence. The indexing function, initialization list and, for arrays, index bound are all optional and may appear in any order in the parameter sequence.

When a table is created using one of these functions, the following sequence of events takes place:

(1) The parameters are sorted out.

(2) If no indexing function is supplied, NULL is taken as the default.

(3) If no initialization list is supplied, an empty list is taken as the default.

(4) If it is an array that is being created and no index bound has been supplied, the index bound to be used is deduced from the initialization list.

(5) If the initialization list is not empty, the initial values are inserted into the table.

(6) The table is returned.

The indexing function must be given as either a procedure or as a name. Not specifying an indexing function is the only way to obtain NULL.

The deduction of index bounds for arrays and the initialization of table values are done by two procedures from the Maple library. The initializations are specified as a list of equations, as a list of values, or as a list of lists. With a list of equations, the left-hand sides are the indices (of the components to be initialized) and the right-hand sides are the values. A list of values may be used to specify the creation of a table or a one-dimensional array, in which case the indices used are consecutive integers starting at 1 (or the lower bound on the indices, if one is given, for an array). For two-dimensional (or higher-dimensional) arrays, the initializations may be specified as a list of the rows of the array, where each row is itself specified as a list (or a list of lists if there are more than two dimensions).

If no index bound is given for an array, then one is deduced from the list of initializations. This is done as follows. If the initialization list is empty, then the array is assumed to be zero-dimensional and the index bound is NULL (i.e. a sequence of zero integer ranges). If the initialization is given as a list of n values, then the array is taken to be one-dimensional and the index bound is the range $1..n$. If the initializations are given as a list of equations, then each range in the index bound is made as restrictive as possible while still encompassing all the indices used in the equations. Finally, if the initialization is given as a list of lists then the number of dimensions is deduced from the level of nesting of the lists and each dimension is indexed from 1.

Examples:

MAPLE Input	Yields

table( );
table([a,b,c]);
table([1=a0, cos(x)=a1]);
	table([ ])†
	table([(1)=a,(3)=c,(2)=b])
	table([(1)=a0,(cos(x))=a1])

array(0..3);
array([x,y,z]);
array([b,c,d], 0..3);
array([NULL=val1]);
array([(2,2)=22, (1,7)=17]);
array(sparse,[5=x,100=y]);
array([[1,2],[3,4]]);
	array(0..3,[ ])
	array(1..3,[(1)=x,(2)=y,(3)=z])
	array(0..3,[(0)=b,(1)=c,(2)=d])
	array([( )=val1])
	array(1..2,2..7,[(2,2)=22,(1,7)=17])
	array(sparse,5..100,[(5)=x,(100)=y])
	array(1..2,1..2,
	[(1,1)=1,(1,2)=2,(2,1)=3,(2,2)=4])

5.2.2. Implicit Table Creation

A table is implicitly created if an assignment is made to an indexed name of the form T[[expression sequence]] when T does not have a previous value. If T does not have a previous value, then the assignment

T[eseq] := expr;

is exactly equivalent to the following statement sequence which uses explicit table creation:

† The indexing function is NULL and no components have been initialized.

```
T := table( ); T[eseq] := expr;
```

This rule is applied recursively if necessary.

Examples:

If A is a table but A[1] has not been assigned, then

```
A[1][2,x] := y;
```

is equivalent to

```
A[1] := table( ); A[1][2,x] := y;
```

If B does not have a previous value, then

```
B[i,k][j] := f(i,j);
```

is equivalent to

```
B := table( ); B[i,k] := table( ); B[i,k][j] := f(i,j);
```

5.3. Evaluation Rules for Tables and Table Components

5.3.1. Evaluation to a Table

Evaluation of an expression in Maple is normally performed in a full, recursive evaluation mode. The result of evaluation is normally the fully evaluated expression. One exception to this rule is when the evaluated expression is a table (or array) and there is an identical exception for a procedure. (See section 6.1 for the evaluation rules for procedures.) Namely, if full evaluation of an expression yields a table structure then the result of evaluation is the *last name evaluated*.

The most common situation is the case of a name T which has been assigned a table (or array) structure, in which case the result of evaluating T will be the name T rather than the value of T. If the value of T is required, the construct

```
op(T)
```

must be used (see the **op** function in section 9.1). In some contexts there will be no name appearing in the evaluation chain, in which case there is no *last name evaluated*; in such a situation, the result of evaluation will be the table structure.

Examples:

```
T[x] := a;    yields   T[x] := a
lprint(T);    yields   table([(x)=a])
f(T);         yields   f(T)
S := T;       yields   S := T
S[x];         yields   a
f(S);         yields   f(T)
op(T);        yields   table([(x)=a])
f(");         yields   f(table([(x)=a]))
```

In particular, note the result of evaluating the expression f(S) above. At that point, S points to the name T which points to the table structure. Thus full evaluation of S leads to the table structure and the result of this evaluation is T because the latter is the *last name evaluated* in the chain of evaluation. Note also the final statement f("). In this case, the evaluation of the function argument leads to a table structure and there is no name in the evaluation chain, so the table structure itself is the result of the evaluation. Of course, the exact same behaviour would occur if the value of T was an array.

5.3.2. Evaluating Table Components

The semantics of referencing a table's components are defined by its indexing function. With the default indexing function, NULL, the usual notion of subscripting is used. With other indexing functions, a procedure determines how indexing is done. This more complicated case is discussed in section 5.5. In this section, the default indexing semantics are described.

Suppose that the value of T is a table with a NULL indexing function. When T[[expression sequence]] is evaluated, the value of the entry in the table is returned, if there is one. If there is not an entry with the [[expression sequence]] as its key, then the table reference "fails" in the following sense. Namely, the value returned is an indexed name where the index is the [[expression sequence]], evaluated, and the zero*th* operand is the *last name evaluated* in the evaluation chain for T.

Examples:

```
t := table( );          yields              t := table([ ])
t[k] := ZZ:             updates table to    table([(k)=ZZ])

s := t:
s[1] := XX:             updates table to    table([(1)=XX,(k)=ZZ])
t[k];                   yields              ZZ
s[1];                   yields              XX
t[2];                   yields              t[2]

j := 2:
t[j];                   yields              t[2]
s[2];                   yields              t[2]

p := proc(a) a[2] end:
p(s);                   yields              t[2]
```

In the above examples, the name 's' evaluates to the name 't', whose value is the table. That is why s[2] yields t[2] when it "fails". In the procedure call, p(s), the name 's' evaluates to the name 't' which is passed in as the argument to the procedure.

Even if the last name in the evaluation chain evaluates to some other object before evaluating to the table, it is still used if a table reference "fails".

Example:

```
a := f(table( ));    yields    a := f(table([ ]))
f := proc(t) print(hello); t end:
a[x];                yields    a[x] after printing "hello"
```

An array is zero-dimensional if its index bound is the null expression sequence. A zero-dimensional array has only one component and the index for this component is NULL.

Example:

```
t := array( );      yields                t := array([ ])
t[ ] := tval:       updates the table to  array([( )=tval])
t[ ];               yields                tval
```

5.3.3. Assigning and Unassigning Table Components

If T is a table and an expression of the form T[[expression sequence]] appears on the left-hand side of an assignment statement, then an entry is assigned in the table. If there was previously no entry in the table with the [[expression sequence]], evaluated, as its key, then a new entry is made. If there already is an entry, then it is updated to reflect the new value.

In many cases it is desired to assign a value to a parameter of a procedure. Table components may be assigned this way, in the same way as names. To assign a table component, an indexed name is passed. Consider the procedure

```
assignsqr := proc(a,b) a := b^2 end
```

So long as the first parameter is a valid left-hand side, the assignment will be made.

Examples:

```
t := table( );
assignsqr(t[2], 4);     assigns the value 16 to t[2]
s := 's';               unassigns s
assignsqr(s[3], 3);     assigns 9 to s[3], implicitly creating a table
```

If the component to be assigned already has a value, then it is necessary to use quotes or the **evaln** function to pass the indexed name rather than its value.

Examples:

After executing the statements

```
t := table( );
for i to 5 do assignsqr(t[i], i) od;
```

assigning new values to the components of 't' may be achieved by

```
for i to 5 do assignsqr(evaln(t[i]), 1/i) od;
```

When the subscript need not be evaluated, quotes may be used:

```
assignsqr('t[1]', x);
```

To make a name stand for itself in Maple, a statement is executed to "unassign" it. Exactly the same thing is done with the components of a table — to remove an entry from a table, it is "unassigned". This may be done either by quoting the right-hand side or by using the **evaln** function.

Examples:

```
a := array([x, y, z]);   yields   a := array(1..3,[(1)=x,(2)=y,(3)=z])
a[1];                    yields   x
a[1] := 'a[1]':
a[1];                    yields   a[1];
lprint(a);               yields   array(1..3,[(2)=y,(3)=z])
i := 3:
a[i] := evaln(a[i]):
lprint(a);               yields   array(1..3,[(2)=y])
```

To determine whether a table component is assigned, the **assigned** function is used. A component T[x] is deemed to be assigned if the evaluation of T[x] would yield a different result than evaln(T[x]). Thus a definition for assigned(T[x]) is:

evalb(T[x] <> evaln(T[x])) .

Examples:

```
a := array(1..3, [e,f]):
a[2];                    yields   f
assigned(a[2]);          yields   true
assigned(a[3]);          yields   false
a := table(sparse):
a[v];                    yields   0
evaln(a[v]);             yields   a[v]
assigned(a[v]);          yields   true
```

5.4. Tables as Objects

5.4.1. Copying Tables

In Maple, only objects of type 'table' may be altered after having been created. This is because it is only with tables that it is valid to make an assignment to a *part* of the object.

The fact that a table object may be altered after creation means that if two names evaluate to the same table, then an assignment to a component of one affects the other as well. To illustrate, if the following statements are executed:

```
a := array([t,x,y,z]);
b := a;
a[1] := 9;
```

then b[1] will evaluate to 9, not 't'. Even if we use the statement

```
b := op(a);
```

in place of the statement

```
b := a;
```

in the above example, we get the same behaviour. In the former case, the name 'b' points to the name 'a' which points to the array structure; in the latter case, 'b' points directly to the array structure as does 'a'. In both cases there is only one array object in existence, and after the assignment a[1] := 9 (or equivalently b[1] := 9) then the value of the first component in the array object will be 9.

For this reason the **copy** function is provided (see section 9.1). It may be used to create a copy of a table upon which operations may be performed without altering the original.

For example, if a procedure makes assignments to components of a table passed as a parameter, then it may be necessary to pass a copy. Suppose that 'decomp' has been assigned a procedure that does an in-place LU matrix decomposition, and takes an array as its only parameter. If it is desired to find the LU decomposition of the matrix given by 'a' while retaining 'a' for further use, then the following statements may be used:

```
b := copy(a);
decomp(b);
```

5.4.2. Tables Local to a Procedure

A variable local to a procedure may be assigned a table, just as it may be assigned an object of any other type. A table object which is created and assigned to a local variable may be returned as the value of the procedure or passed out through one of the parameters, in a manner similar to any other expression. The main distinction with tables is that a name whose value is a table evaluates to its own name, not to the object. Therefore, it is necessary to use the **op** function if the table object is to be obtained. In particular, if a local variable 'T' has been assigned a table and this table is to be returned from the procedure then one must be careful to return the value op(T) rather than simply T. (Returning a local variable name out of a procedure is very unwise.)

Example:

```
# Put the coefficients of a polynomial in a table.
getcoeffs := proc(poly, var)
        local Cs, c, i;
        if not type(poly, polynom, var) then
            ERROR(`must have a polynomial`)
        fi;
        Cs := table( );
        for i from ldegree(poly, var) to degree(poly, var) do
            c := coeff(poly, var, i);
            if c <> 0 then Cs[i] := c fi
        od;
        op(Cs)
end;
```

```
Cs := table([this, that]);      yields   Cs := table([(1)=this,(2)=that])
getcoeffs(3*x^67 + y, x);       yields   table([(0)=y,(67)=3])
lprint(Cs);                     yields   table([(1)=this,(2)=that])
```

5.4.3. Tables as Parameters

A table may be passed as a parameter into or out of a procedure. Components added to the table or removed from the table while the procedure is executing affect the globally visible table, since it is the same object.

Passing an unnamed table object as a parameter may lead to awkward results if components which do not have values are used. If the procedure makes a component reference that "fails" and assigns it to another component of the same table, then doing the assignment creates a self-referential data structure. (Just as doing x := y; y := 'x^2' does.) This would lead to an infinite evaluation recursion the next time the component was evaluated. It is recommended to always pass, as a parameter, the name of a table (say A) rather than the table structure (op(A) if the value of A is a table structure).

5.4.4. Automatic Loading of Tables

It is possible to define large tables that get loaded only when a component is referenced. This is done in the same way that procedures can be made to be read in only when used.

To cause a table to be loaded automatically, it is assigned an unevaluated call to **readlib**. If a user wants T to be loaded when it is used, then he makes the assignment

```
T := 'readlib('T', filename)';
```

where 'filename' is the name of the file in which the table has been saved. (See section 6.9 for a discussion of the **readlib** function.)

Suppose a user enters Maple and executes the following statements:

```
Linverse := table( );
Linverse[1/s^n] := t^(n-1)/(n-1)!;
Linverse[1/(s^2 + a^2)] := sin(a*t)/a;
save `/u/jqpublic/laplace.m`;
quit
```

If in a subsequent Maple session the assignment

```
Linverse := 'readlib('Linverse', `/u/jqpublic/laplace.m`)';
```

has been made, then evaluating

```
Linverse[1/s^n]
```

causes **readlib** to be executed and the table is read in. The selection operation then evaluates to

```
t^(n-1)/(n-1)!
```

5.5. Indexing Functions

5.5.1. The Purpose of Indexing Functions

The semantics of indexing into a table are described by its indexing function. Using an indexing function, it is possible to do such things as efficiently store a symmetric matrix or count how often each element of a table is referenced. Because each table defines its own indexing method, generic programs can be written that do not need to know about special data representations. For example, the same function could be used to perform an operation on sparse matrices as for dense matrices.

The normal method of indexing, described in section 5.3, is used when the indexing function of a table is NULL. The semantics correspond roughly to those of common programming languages, with the added notion of "failing" if a component has not been assigned.

If the indexing function for a given table is not NULL, then all indexing into that table is done through a procedure. This procedure is invoked whenever a selection operation of the form T[[expression sequence]] is encountered and the value of T is the table.

Three parameters are passed to the procedure:

(i) the name of the object which is being indexed, T,

(ii) a list containing the index, [[expression sequence]],

(iii) a boolean value which is *true* (*false*) if the expression is being evaluated as on a left-hand (right-hand) side of an assignment.

The value returned by the procedure is used in the place of the specified selection operation.

The indexing function may be the procedure itself, or a name. Certain names are known to the basic system as built-in indexing functions. If a name is given which is not one of these, a function call is made using `index/` . [[name]] . First the current session environment is searched for this name. If it is not found, the Maple system library is searched for the file `` . libname . `/index/` . [[name]] . `.m` . If no such file exists, then `index/` . [[name]] is applied as an undefined function.

5.5.2. Indexing Functions Known to the Basic System

At present, three names are known to the basic Maple system as indexing functions. These are *symmetric*, *antisymmetric*, and *sparse*. Three additional indexing procedures are currently defined in the Maple library (and hence act as names "known" to the Maple system): *diagonal*, *identity*, and *sparsesymm*.

The indexing function *symmetric* is used for tables in which the value of a component is independent of the order of the expressions in the index. The most common application is for symmetric matrices. When a component of a table with this indexing function is referenced, the index expression sequence is re-ordered to give a unique key. (The sort is done using the same internal ordering as for sets.)

Examples:

```
A := array(1..10,1..10,symmetric); yields
                A := array(symmetric,1..10,1..10,[ ])
A[1,2];                          yields A[1,2]
A[2,1];                          yields A[1,2]
A[i,j] - A[j,i];                 yields 0
A[3,4] := x;                     yields A[3,4] := x
A[4,3] := y;                     yields A[3,4] := y
op(A);                           yields
            array(symmetric,1..10,1..10,[(3,4)=y])
T := table(symmetric);           yields T := table(symmetric,[ ])
T[function,continuous,odd] := f; yields T[function,continuous,odd] := f
T[odd,continuous,function] := f; yields T[function,continuous,odd] := f
```

The *antisymmetric* indexing function yields the result of *symmetric*, multiplied by -1 if all components of the index are different and an odd number of transpositions were necessary to re-order the index. If two or more components of the index are the same, *antisymmetric* returns 0.

Examples:

```
B := table(antisymmetric); yields B := table(antisymmetric,[ ])
B[i,j];                    yields B[i,j]
B[j,i];                    yields -B[i,j]
B[i,j,k] + B[i,k,j];       yields 0
B[i,k,k];                  yields 0
B[i,j] := v;               yields B[i,j] := v
B[j,i] := u;               yields Error, Illegal use of an object as a name
```

The indexing function *sparse* is used with tables for which a component is assumed to have value 0 if it has not been explicitly assigned. Suppose T is a table with this indexing function. Evaluating T[[expression sequence]] on a right-hand side yields the component's value, if it has been assigned, or 0, if it hasn't. When T[[expression sequence]] is evaluated on a left-hand side, the indexing function always returns the indexed name T[[expression sequence]] . (Returning 0 would make assigning components impossible.)

Examples:

```
U := array(1..100,sparse,[90=u1]);                    yields
                U := array(sparse,1..100,[(90)=u1])
V := array(1..100,sparse,[34=v1]);                    yields
                V := array(sparse,1..100,[(34)=v1])
s := 0;
for i to 100 do
    s := s + U[i] + V[i]
od;
s;                                                    yields u1 + v1
```

The three indexing functions defined in the Maple library have the following definitions. The indexing function *diagonal* returns zero as the value of all table components for which the indices are not identical. The indexing function *identity* returns the value one if the indices are all identical, and returns zero otherwise. The indexing function *sparsymm* is used to define a table which is both sparse and symmetric (see the individual definitions above).

5.5.3. User-Defined Indexing Functions

A user may create his own indexing function by writing a procedure which returns the expression to be used, given the object being indexed, the index, and an indication of whether a left- or right-hand side is desired.

Suppose we wish to define a large tridiagonal matrix. To avoid storing the off-diagonal elements, the following procedure may be used as the indexing function:

```
t3 := proc(A, index, is_lhs) local dummy;
            op(1,index) - op(2,index);
            if not type(", integer) or abs(") <= 1 then
                subs(dummy = op(index), 'A[dummy]')
            else
                0
            fi
end;
```

The array may be created by the statement

```
Tri := array(op(t3), 1..10000, 1..10000);
```

or by the statements

```
`index/tridiagonal` := t3;
Tri := array(tridiagonal, 1..10000, 1..10000);
```

In the first case, 'Tri' would have the procedure as its first operand. In the second, it would have the name 'tridiagonal'. Assume for the following discussion, that 'Tri' has been assigned by the second method.

To explain how this procedure works, suppose the statement

```
Tri[3,20] := rhs;
```

is executed. After the right-hand side has been evaluated, 'Tri' is evaluated and found to be an array. Next, the index is evaluated and found to be within bounds. The indexing function is then found to be 'tridiagonal' so the following procedure call is made:

```
`index/tridiagonal`('Tri', [3,20], true)
```

The third parameter indicates that the evaluation is being done for the left-hand side of an assignment. (In this case the procedure `index/tridiagonal` does not use the third parameter, but it is passed anyway.) The element referred to is found not to be on the tri-diagonal band so the *else* part is executed and the value 0 is returned. Since it is impossible to make an assignment to 0, the assignment statement generates an error message. This is reasonable; it should not be possible to make assignments to the off-band entries of a tridiagonal matrix.

If the statement

```
Tri[99,100];
```

is executed, then the following events occur. As before, 'Tri' is evaluated to a table and the index is found to be within bounds. Then the procedure call

```
`index/tridiagonal`('Tri', [99,100], false)
```

is made. Since this component is found to be on the upper diagonal, the statement

```
subs(dummy = op([99,100]), 'Tri[dummy]')
```

is executed. This returns Tri[99,100], unevaluated, as the procedure value. Then, if Tri[99,100] has been assigned, its value is retrieved. Otherwise the table reference fails as usual.

It is important to avoid evaluating the expression T[99,100] accidentally inside the procedure, as this would cause an infinite recursion. This is the reason that **subs** was used to create the expression returned by `index/tridiagonal`.

As a second example, suppose we want to count the number of assignments made to components of various arrays and other tables. The counts will be kept in a table, 'Count_table', initialized by

```
Count_table := table(sparse);
```

(Note that Count_table needs no explicit initialization of values.) If 'A' is one of the tables to be monitored and an assignment is made to A[1,2], then Count_table[A,[1,2]] will be incremented by one.

The procedure below may be used as the indexing function for the tables to be monitored:

```
`index/count` := proc(T, index, is_lhs)
        local dummy;
        if is_lhs then
            Count_table[T, index] := Count_table[T, index] + 1
        fi;
        subs(dummy = op(index), 'T[dummy]')
end;
```

Then, the tables under investigation are created as follows

```
t1 := table(count);
aa := array(1..100, count);
```

and used normally.

For a third example, we consider the "Riemann tensor" from general relativity. For our purposes it may be considered to be an array with (0..3,0..3,0..3,0..3) as its index bound. This object would have 256 components if all were independent. However, the Riemann tensor has (among others) the following symmetry properties

```
R[i,j,k,l] = -R[j,i,k,l]
R[i,j,k,l] = -R[i,j,l,k]
R[i,j,k,l] =  R[k,l,i,j] .
```

These imply that at most 21 components are algebraically independent. The array could be created with the following procedure as its indexing function:

```
`index/Riemann` := proc(A,ix)
        local i,j,k,l,dummy;
        option remember;
        i := op(1,ix); j := op(2,ix); k := op(3,ix); l := op(4,ix);
        if i = j or k = l then  0
        elif not order(i,j) then  -A[j,i,k,l]
        elif not order(k,l) then -A[i,j,l,k]
        elif not order([i,j],[k,l]) then A[k,l,i,j]
        else subs(dummy = op(ix), 'A[dummy]')
        fi
end;
```

where 'order' is a boolean function defining an ordering on expressions, such as

```
order := proc(a,b) evalb( a = op(1,{a,b}) ) end
```

(which uses the ordering defined by Maple's ordering of elements in a set). The procedure `index/Riemann` is recursive, since evaluating the expressions -A[j,i,k,l], -A[i,j,l,k], and A[k,l,i,j] causes the indexing function to be called again.

6. PROCEDURES

6.1. Procedure Definitions

One instance of an expression in Maple is a procedure definition, which has the general form

proc ([[nameseq]]) local [[nameseq]]; options [[nameseq]]; [[statseq]] end .

A procedure definition is normally assigned to a [[name]] and then it may be invoked using the syntax

[[name]] ([[exprseq]]) .

When a procedure is invoked the statements in [[statseq]] are executed sequentially (and some of the names have special semantics as described below). The *value* of a procedure invocation is the value of the last statement in [[statseq]] that is executed. For example,

```
f := proc (x) x^2 end;   yields   f := proc (x) x^2 end
f(2);                    yields   4
```

Evaluation of an expression in Maple is normally performed in a full, recursive evaluation mode. The result of evaluation is normally the fully evaluated expression. One exception to this rule is when the evaluated expression is a procedure definition, just as the case of tables is an exception. (See section 5.3 for the evaluation rules for tables.) Namely, if full evaluation of an expression yields a procedure definition then the result of evaluation is the *last name evaluated*. The most common situation is the case of a name f which has been assigned a procedure definition, in which case the result of evaluating f will be the name f rather than the value of f. If the value of f is required, the construct

op(f)

must be used (see the **op** function in section 9.1). For example:

```
f := proc (x) x^2 end:
f;                              yields   f
op(f);                          yields   proc (x) x^2 end
apply := proc (g,a) g(a) end:
apply(f, t);                    yields   t^2
apply(op(f), t);                yields   t^2
h := f;                         yields   h := f
h(3);                           yields   9
```

It is possible in Maple to define and invoke a procedure without ever assigning the procedure definition to a ⟦name⟧. For example:

```
proc (x) x^2 end;        yields   proc (x) x^2 end
%(2);                    yields   4
proc (x) x^2 end (3);    yields   9
```

The result of evaluating the first statement above is that the procedure definition is stored and it is available immediately afterwards as the value of the ditto operator. In the second statement, the ditto operator returns the procedure definition as its value and the procedure is then invoked with the argument 2. The third statement is similar to the second, but it expresses the procedure definition and its invocation in a single statement.

Another example of using a procedure definition without a name is when a simple procedure is to be *mapped* onto an expression, as in the following example:

```
a := [1, 2, 3, 4, 5];              yields   a := [1, 2, 3, 4, 5]
map( proc (x) x^2 end, a );        yields   [1, 4, 9, 16, 25]
```

(See the **map** function in section 9.1.)

The keywords 'proc' and 'end' may be viewed as brackets which specify that the ⟦statseq⟧ is to remain unevaluated when the procedure definition is evaluated as an expression. The simplest instance of a procedure definition involving no formal parameters, no local variables, and no options can be seen in the following definition of a procedure called max:

```
max := proc ( ) if a>b then a else b fi end .
```

Executing the statements

```
a := 25/7;  b := 525/149;  max( );
```

yields 25/7 as the value of the procedure invocation max( ). This procedure is making use of the names 'a' and 'b' as *global names*. In Maple, all names are global names unless otherwise specified. One instance of non-global names is the case of *formal parameters* which are specified within the parentheses immediately following the keyword 'proc'. A more useful definition of the above procedure max can be obtained by making the names 'a' and 'b' formal parameters:

```
max := proc (a,b) if a>b then a else b fi end
```

This procedure may now be invoked in the form max(expr1, expr2) where expr1 and expr2 are expressions. For example, max(25/7, 525/149) evaluates to 25/7. The names 'a' and 'b' are now local to the procedure, so that if these names have values external to the procedure the external values

neither effect, nor are affected by, the invocation of the procedure.

6.2. Parameter Passing

The semantics of parameter passing are as follows. Suppose the procedure invocation is of the form

$$name\ (\ expr_1, expr_2, \ldots, expr_n\ )\ .$$

First, *name* is evaluated and let us suppose for now that it evaluates to a procedure definition with formal parameters

$$param_1, param_2, \ldots, param_n\ .$$

Next, the *actual parameters* $expr_1, \ldots, expr_n$ are evaluated in order from left to right. Then every occurrence of $param_i$ in the ⟦statseq⟧ which makes up the body of the procedure is substituted by the value of the corresponding actual parameter $expr_i$. It is important to note that these parameters will not be evaluated again during execution of the procedure body. (The consequences of this fact are explained in section 6.6 below.) In terms of traditional parameter passing mechanisms used by various computer languages, Maple's parameter passing could be termed 'call by evaluated name'. In other words, all actual parameters are first evaluated (as in 'call by value') but then a strict application of the substitution rule is applied to replace each formal parameter by its corresponding actual parameter (as in 'call by name').

It is possible for the number of actual parameters to be either greater than, or less than, the number of formal parameters specified. If there are too few actual parameters then a semantic error will occur if (and only if) the corresponding formal parameter is referenced during execution of the procedure body. The case where the number of actual parameters is greater than the number of specified formal parameters is, on the other hand, fully legitimate. If an actual parameter is never referenced it is simply ignored (but it will be evaluated).

Maple allows an alternate mechanism for referencing parameters within a procedure body; namely, via the special name 'args' which has as its value the expression sequence of actual parameters with which the procedure was invoked. Thus the selection operation 'args[i]' references the i^{th} actual parameter. For example, the above procedure max could be defined without any specified formal parameters as follows:

```
max := proc ( ) if args[1] > args[2] then args[1] else args[2] fi end .
```

This procedure may now be invoked exactly as before with two actual parameters and the semantics are identical to the previous definition.

In addition to the special name 'args', there are two other special names that Maple understands within a procedure body: 'nargs' and 'procname'. The value of the name 'nargs' is the number of actual parameters (i.e., the number of arguments) with which the procedure was invoked. The value of the name 'procname' is the procedure name in the sense of the *last name evaluated* (see section 6.1). If there is no name associated with the procedure then 'procname' has as its value the name 'unknown'. (See section 6.7 for an example of the use of 'procname'.)

As an example of the use of the name 'nargs', let us generalize our procedure max so that it will be defined to calculate the maximum of an arbitrary number of actual parameters. Consider the following procedure definition:

```
max := proc ( )
            result := args[1];
            for i from 2 to nargs do
                  if args[i] > result then result := args[i] fi
            od;
            result
end;
```

With this definition of max we can find the maximum of any number of arguments. Some examples are:

```
max( 25/7, 525/149 );          yields   25/7
max( 25/7, 525/149, 9/2 );     yields   9/2
max( 25/7 );                   yields   25/7
max( );                        yields   Error, (in max) improper op or
                                                subscript selector
```

where the latter case is an example of a procedure being called with too few actual parameters. If we wish to change our definition of max so that the procedure invocation max( ) with an empty parameter list will return the null value then we may check for a positive value of nargs in a selection statement as in the following definition of max.

```
max := proc ( )
        if nargs > 0 then
                result := args[1];
                for i from 2 to nargs do
                        if args[i] > result then result := args[i] fi
                od;
                result
        fi
end;
```

6.3. Local Variables

The mechanism for introducing *local variables* into a Maple procedure is to use the 'local part' of a procedure definition. The 'local part' must appear immediately following the parentheses enclosing the formal parameters, and its syntax is

```
local ⟦nameseq⟧;
```

The semantics are that the names appearing in ⟦nameseq⟧ are to be local to the procedure. In other words, this can be viewed as causing a syntactic renaming of every occurrence of the specified names within the procedure body.

As an example, let us reconsider the latest definition of max appearing above. There are two global variables appearing in the procedure definition which we would almost certainly want to make local: result and i. This is effected by the following version of the procedure definition.

```
max := proc ( )
        local result, i;
        if nargs > 0 then
                result := args[1];
                for i from 2 to nargs do
                        if args[i] > result then result := args[i] fi
                od;
                result
        fi
end;
```

During the execution of a procedure body, the evaluation of local variables is performed in a *one-level evaluation* mode. This is in contrast to variables which are global to the Maple session, for which full recursive evaluation is applied. The restricted evaluation mode for local variables leads to greater efficiency and the user will find that this evaluation mode is

normally sufficient. For exceptions where full recursive evaluation is required, see the **eval** function in section 9.1.

6.4. Options

There is a facility to specify *options* for a procedure by using the 'options part' of a procedure definition. The 'options part' must appear immediately after the 'local part' and its syntax is either of the following two forms:

option [[nameseq]];
options [[nameseq]];

There are currently five names which are recognized by Maple when specified in the 'options part' of a procedure definition: *remember*, *builtin*, *system*, *operator*, and *trace*.

Option *remember* indicates that the result of any invocation of the procedure is to be stored in a *remember table*. See section 6.5 below for a detailed discussion of remember tables in Maple.

Option *builtin* is used to identify Maple's built-in functions. For example, the **type** function is a built-in function in Maple and the result of displaying the value of the name 'type' is:

op(type); yields proc ( ) options builtin; 85 end

The 'builtin' option identifies this as a built-in function rather than a Maple procedure in the external library, and the number 85 is a special number uniquely identifying the particular **type** function.

Option *system* serves to identify procedures which are considered to be "system functions", meaning functions for which entries may be deleted from the remember table at garbage collection time. If this option is not specified for a procedure having option remember, all entries in the remember table will "survive" garbage collections.

Option *operator* identifies the procedure to be a *functional operator*. (See chapter 7 for a discussion of functional operators in Maple.)

Option *trace* is used to specify that run-time tracing of the procedure should be displayed during execution. The user may explicitly place option 'trace' on a procedure to be traced. However, there is a user-interface function called **trace** (see section 9.1) which is often more convenient for this purpose.

6.5. Remember Tables

With every Maple procedure, there may be associated a *remember* table. A remember table is a hash table in which the arguments to a procedure call are used as the table index, and the result of the procedure call is used as the table value. When a Maple procedure is invoked, it's remember table is searched on the actual arguments. If the actual arguments are found as an index in the remember table, the corresponding value is returned as the result of the procedure call. Otherwise the body of the procedure is executed to compute the result. The purpose of remember tables, therefore, is to make use of fast table lookup to avoid expensive recomputation. This is effective when function values are both relatively expensive to compute and likely to be recomputed.

The motivation for remember tables comes from the observation that many operations in symbolic computation reproduce subexpressions many times in their results. Any further operations on these results must deal with these common subexpressions repeatedly. By storing the result for each subexpression in the remember table, we avoid doing any recomputation. For example, consider the operation:

```
taylor( sin(a+x), x=0 );
                                        2                       3
sin(a) + cos(a) x + (-1/2 sin(a)) x  + (-1/6 cos(a)) x

                    4                       5       6
   + 1/24 sin(a) x  + 1/120 cos(a) x  + O(x )
```

Notice that the resulting series contains many occurrences of the expressions `sin(a)` and `cos(a)`. Subsequent operations on the resulting series expression will repeatedly encounter these subexpressions.

Values can be entered into a remember table in two ways. First, automatically, by specifying the 'remember' option in the options clause of the procedure definition. Second, explicitly, by means of a 'function assignment'.

By specifying the 'remember' option in a procedure definition, the system automatically records the result of the procedure invocation in the remember table just before returning from the procedure. That is, the actual argument(s) together with the result are inserted as the table index and table value respectively into the remember table.

The 'function assignment' allows one to selectively enter results into a remember table. It closely resembles assignment of a table component (see section 5.3.3). It has the form

```
f(x) := y
```

where *f* must evaluate to a procedure. The meaning of this assignment is as follows. First, the name of the function *f* is evaluated, then the argument(s) *x* are evaluated. Next, the right hand side *y* is evaluated. Finally, the table entry (*x*,*y*) is then inserted into *f*'s remember table and the value of *y* is returned as the result of the assignment. Note that the function call *f(x)* is not executed.

Values may also be removed from remember tables. The purpose of removing values from a remember table is to free the space that the remember table occupies, and to allow the objects in the remember table to be collected by the garbage collector. If we could not remove entries from a remember table, the problem of running out of memory space could be serious. Entries can be removed from remember tables either explicitly or automatically, as follows. (See also the **forget** function in section 9.2.)

A procedure's remember table can be accessed as the 4th operand of the procedure. One can delete the entire remember table by substituting the NULL expression in place of it. Alternatively, one can point to it by assigning it to a variable. When accessed in this way, the remember table can be manipulated as a user-level Maple table (see section 8.3.5). For example, if *f* is assigned to a procedure, the statements

```
T := op(4,op(f));
T[1] := evaln(T[1]);
```

remove the value for *f(1)* from *f*'s remember table. (See the **evaln** function in section 9.1.)

By specifying the 'system' option in the procedure definition, values are removed from the remember table automatically by the system during garbage collection. To be precise, the values removed are those for which either the argument or the result is not a "live" structure.

Remember tables have proven to be a very useful language feature. Although initially introduced to avoid repeated computations, remember tables have also been used as a programming tool. The remember option allows the coding of linear recurrences and other recursive formulae directly, without a loss of efficiency. For example, the following procedure to compute Fibonacci numbers:

```
fib := proc(n)
    if n < 2 then n else fib(n-1) + fib(n-2) fi
end
```

takes exponential time to compute, whereas

```
fib := proc(n) option remember;
    if n < 2 then n else fib(n-1) + fib(n-2) fi
end
```

takes only linear time. Of course, the same improvement could have been obtained either by recoding the procedure to explicitly make use of an array, or by computing the values in a loop. The advantage of option remember is that it allows the function to be coded in the most natural manner. The resulting code is very easy to read because the algorithmic intent is not obscured by code for saving intermediate results. A slightly faster version of the Fibonacci program can be obtained by explicitly stating the base of the recursion as follows:

```
fib := proc(n) option remember; fib(n-1) + fib(n-2) end;
fib(0) := 0;
fib(1) := 1;
```

Remember tables can be used to tabulate special or exceptional cases in a succinct way by using the functional assignment. They provide direct support for the notion of 'dynamic' programming. In coding complicated recursive algorithms, one can use remember tables to exploit the occurrence of repeated computations, sometimes resulting in a dramatic improvement in running time.

A note of caution. Remember tables should not be used in the presence of side effects. Suppose that one writes a Maple procedure to compute floating-point values for the function *f* to arbitrary precision. Specifying option remember would be incorrect in this case because the precision to which the floating-point values are computed is controlled by the global variable Digits. For example, suppose that the value of f(1) is .12345678901234567890 at 20 digits precision. Consider the following:

```
Digits := 10;
f(1);
                    .1234567890
Digits := 20;
f(1);
                    .1234567890
```

In the second call to *f*, even though the correct value is .12345678901234567890, the value returned was .1234567890 because

.1234567890 was found in *f*'s remember table.

The moral is to beware of applying option remember with procedures which exploit side effects.

6.6. Assigning Values to Parameters

Let us now consider an example of a procedure where we may wish to return a value into one of the actual parameters. Recall that the integer quotient q and the integer remainder r of two integers a and b must satisfy the 'Euclidean division property'

a = b*q + r

where either r = 0 or abs(r) < abs(b). This property does not uniquely define the integers q and r, but let us impose uniqueness by choosing

q = trunc(a/b)

using the built-in Maple function **trunc**. The remainder r is then uniquely specified by the above Euclidean division property. (Note: This choice of q and r can be characterized by the condition that r will always have the same sign as a.) The following definition of the procedure 'rem' returns as its value the remainder after division of the first parameter by the second parameter, and it also returns the quotient as the value of the third parameter (if present).

```
rem := proc (a,b,q)
            local quot;
            quot := trunc(a/b);
            if nargs > 2 then q := quot fi;
            a - quot * b
end;
```

The procedure rem as defined here may be invoked with either two or three parameters. In either case the value of the procedure invocation will be the remainder of the first two parameters. The quotient will be returned as the value of the third parameter if it appears. At this point recall that the semantics of parameter passing specify that the actual parameters are evaluated and then substituted for the formal parameters. Therefore, an error will result if an actual parameter which is to receive a value does not evaluate to a valid name. It follows that when a name is being passed into a procedure for such a purpose it should usually be explicitly quoted (to avoid having it evaluated to some value that it may have had previously). The following statements will illustrate the point.

```
rem( 5, 2 );          yields   1
rem( 5, 2, 'q' );     yields   1
q;                    yields   2
rem( –8, 3, 'q' );    yields  –2
q;                    yields  –2
rem( 8, –3 );         yields   2
rem( 8, 3, q );       yields   Error, (in rem) Illegal use of a formal
                                  parameter
```

The latter error message arises because the actual parameter q has the value –2 from a previous statement, and therefore the value –2 is substituted for the formal parameter q in the procedure definition yielding an invalid assignment statement. The solution to this problem is to change the actual parameter from q to 'q'.

When values are assigned to parameters within a procedure, a restriction which must be understood is that parameters are evaluated only once. Basically this means that formal parameter names cannot be freely used like local variables within a procedure body, in the sense that once an assignment to a parameter has been made that parameter should not be referred to again. The only legitimate purpose for assigning to a parameter is so that on return from the procedure the corresponding actual parameter has been assigned a value. As an illustration of this restriction, consider a procedure get_factors which takes an expression expr and, viewing it as a product of factors, determines the number n of factors and assigns the various factors to the names t[1], t[2], . . . , t[n] . Here is one attempt at writing a procedure for this purpose.

```
get_factors := proc (expr,t,n)
            local i;
            if type(expr, `*`) then
                  n := nops(expr);
                  for i to n do t[i] := op(i,expr) od
            else
                  n := 1;
                  t[1] := expr
            fi
end;
```

If this procedure is invoked in the form

```
get_factors(x*y, 't', 'number');
```

the result is 'Error, (in get_factors) unable to execute for statement'. What has happened is that the third actual parameter is a name (as it must be because it is to be assigned a value within the procedure) and when

execution reaches the point of executing the for-statement, the limit n in the for-statement is the name 'number' that was passed in. The point is that the formal parameter n is evaluated only once upon invocation of the procedure and it will not be re-evaluated. A general solution to this type of problem is to use local variables where necessary, and to view the assignment to a parameter as an operation that takes place just before returning from the procedure. For our example, the following procedure definition uses this point of view.

```
get_factors := proc (expr,t,n)
        local i, nfactors;
        if type(expr, `*`) then
        nfactors := nops(expr);
            for i to nfactors do t[i] := op(i,expr) od
        else
            nfactors := 1;
            t[1] := expr
        fi;
        n := nfactors
end;
```

Another solution to the problem in this example is to explicitly invoke the **eval** function (see section 9.1). The following procedure definition uses this approach.

```
get_factors := proc (expr,t,n)
        local i;
        if type(expr, `*`) then
            n := nops(expr);
            for i to eval(n) do t[i] := op(i,expr) od
        else
            n := 1;
            t[1] := expr
        fi
end;
```

6.7. Error Returns and Explicit Returns

The most common return from a procedure invocation occurs when execution 'falls through' the end of the ⟦statseq⟧ which makes up the procedure body, and the value of the procedure invocation is the value of the last statement executed. There are two other types of returns from procedures.

An *error return* occurs when the special function call

ERROR(⟦exprseq⟧)

is evaluated. This function call normally causes an immediate exit to the top level of the Maple system and the error header

Error, (in ⟦procname⟧)

is printed out, followed by the evaluated sequence of expressions which are parameters to the ERROR function call (typically a single string). Here, ⟦procname⟧ refers to the name of the procedure in which the ERROR return occurred, in the sense of the *last name evaluated* (see section 6.1). If there is no name associated with the procedure then the name 'unknown' is printed in place of ⟦procname⟧. Errors can be *trapped* by use of the **traperror** function and the global variable **lasterror** stores the value of the latest error message. (See **traperror** in section 9.1).

An *explicit return* occurs when the special function call

RETURN(⟦exprseq⟧)

is evaluated. This function call causes an immediate return from the procedure and the value of the procedure invocation is the value of the ⟦exprseq⟧ given as the argument in the call to RETURN. In the most common usage ⟦exprseq⟧ will be a single ⟦expression⟧ but a more general expression sequence (including the null expression sequence) is valid. It is an error if a call to the function RETURN occurs at a point which is not within a procedure definition.

A particular form of explicit return is often used as a "fail return", in the sense that the computation cannot be carried out and it is desired to return the unevaluated function invocation as the result. For this purpose, the two special names 'procname' and 'args' are used, as follows:

RETURN('procname (args)')

Recall that the value of 'procname' is the name of the procedure currently being executed and the value of 'args' is the expression sequence of arguments with which the current procedure was invoked. (See section 6.2.) The function invocation appearing here must be quoted to prevent evaluation, for otherwise an infinite recursion will occur.

As an example of a procedure which includes an error return and a fail return, let us consider one final procedure definition for the function 'max'. The latest definition that we developed in section 6.3 for the function 'max' has a property which makes it unacceptable as a general library function for a computer algebra system, for example. Namely, if a user calls this function with an argument that does not evaluate to a numeric constant, such as

in max(x, y) where x and y have not been assigned any values, then the result is an error message from the Maple system: 'Error, (in max) cannot evaluate boolean'. This error results from attempting to execute an if statement of the form

```
if y>x then . . .
```

where x and y are indeterminates, but the error message is not very informative to the user. In order to improve this situation, type-checking should be done within the 'max' procedure and appropriate action should be taken if an invalid argument is encountered. The following procedure definition shows how this can be done using the "fail return" form of an explicit RETURN, so that the result of the function call max(x, y) will be the unevaluated function max(x, y). This procedure also shows the use of the ERROR return facility to return an error message if the function is called with no arguments.

```
max := proc ( )
        local result, i;
        if nargs = 0 then
                ERROR(`no arguments specified`)
        else
                result := args[1];
                for i to nargs do
                        if not type( args[i], numeric ) then
                                RETURN('procname ( args )')
                        fi;
                        if args[i] > result then result := args[i] fi
                od;
                result
        fi
end;
```

6.8. Boolean Procedures

The names *true* and *false*, which are defined to be constants in Maple, may be freely manipulated as names (but they may not be assigned values). It follows that boolean procedures may be written like any other procedures. As an example of a boolean procedure, consider the following definition of a function called 'member' which tests for list membership.

```
member := proc (element, l)  local i;
            false;
            for i to nops(l) while not " do
                  element = op(i, l)  od;
            evalb(")
end;
```

Some examples invoking this procedure follow.

```
member( x*y, [1/2, x*y, x, y] );    yields   true
member( x, [1/2, x*y] );            yields   false
member( x, [ ] );                   yields   false
```

In the above procedure, note the use of the nullary operator " in the while-part of the loop to refer to the 'latest expression'. Alternatively, this could be coded with the use of another local variable. Also note the use of the **evalb** function in the final statement. If the **evalb** function was not applied then the value returned from this procedure would be an algebraic equation:

```
element = op(i, l)
```

rather than a boolean value. During execution of the loop, this equation appears (as the value of ") in a boolean context and is automatically evaluated as a boolean. But outside of this boolean context, it is necessary to force evaluation via the **evalb** function.

6.9. Reading and Saving Procedures

It is usually convenient to use a text editor to develop a procedure definition and to write it into a file. The file can then be read into a Maple session. For example, suppose that the max procedure is written into a file with filename max (more specifically, the filename on the host computer corresponding to Maple's filename max – see the appendices for system-specific information). In a Maple session the statement

```
read max;
```

will read in the procedure definition. Since this procedure is in 'user format' Maple will echo the statements as they are read in. Once the procedure is debugged, it is desirable to save it in 'Maple internal format' so that whenever it is read into a Maple session the reading is very fast (and no time is spent displaying the statements to the user). To accomplish this, one must use a Maple filename ending in the characters '.m'. Within Maple the 'user format' file is read in and then Maple's save statement is used to save the file in 'Maple internal format'. For example, suppose that

we have saved our procedure definition as discussed above. If we enter the Maple system and execute the statements

```
read max;
save `max.m`;
```

we will have saved the internal representation of the procedure in the second file. This file may be read into a Maple session at any time in the future by executing the statement

```
read `max.m`;
```

which will update the current Maple environment with the contents of the specified file. (It is often convenient to place the 'save' statement at the end of the 'user format' file so that simply reading in the file will cause it to be saved in 'Maple internal format'.)

A special case of reading procedure definitions in 'Maple internal format' can be accomplished using the built-in function **readlib**. Specifically, the function invocation readlib(pname) will cause the following Maple **read** statement to be executed:

```
read `` . libname . `/` . pname . `.m`
```

where 'libname' is a global name in Maple which is initialized to the pathname of the Maple system library on the host system. The value of 'libname' on any host system can be determined by entering Maple and simply displaying its value. The filename being specified here is a Maple filename which, in general, would be translated into an appropriate filename on the host system. The **convert** function may be used in the form

```
convert( ⟦Maple filename⟧, hostfile )
```

to cause Maple to display the actual filename that will be used on the host system.

The complete Maple filename being specified in the above **read** statement is a concatenation of the values of 'libname', '/', 'pname', and the suffix '.m', which could alternatively be specified by

```
cat( libname, `/`, pname, `.m` ) .
```

In order to specify this concatenation using only Maple's concatenation operator '.' it is necessary to concatenate these values to the null string ``, because the left operand of Maple's concatenation operator is not fully evaluated but is simply evaluated as a name. (See section 3.2.2.)

The **readlib** function has a more general functionality than was indicated above. If it is called with two arguments then the second argument is the complete pathname of the file to be read, and the first argument

'pname' is the procedure name which is to be defined by this action. Thus the following two function calls are equivalent:

```
readlib( 'f' )
readlib( 'f', `` . libname . `/f.m` )
```

but if the procedure definition for 'f' is not in the standard Maple system library then the second argument is required to specify the correct file. (Even more generally, the **readlib** function can be called with several arguments in which case all arguments after the first are taken to be complete pathnames of files to be read, and the first argument is a procedure name which is to be defined by this action.) The definition of the **readlib** function involves more than just the execution of one or more **read** statements. This function will also check to ensure that after the files have been read, 'pname' has been assigned a value and this value is returned as the value of the **readlib** function. In other words, the **readlib** function is to be used when the purpose of the **read** is to define a procedure named 'pname' (and some other names may or may not be defined at the same time).

A common application of the **readlib** function is to cause automatic loading of files. For this purpose, the value of 'pname' is initially defined to be an unevaluated **readlib** function, as in one of the following assignments:

```
pname := 'readlib( 'pname' )';
pname := 'readlib( 'pname', filename )';
```

where the single quotes around the argument 'pname' are required to avoid a recursive evaluation. Then if there is subsequently a procedure invocation pname(...), the evaluation of 'pname' will cause the **readlib** function to be executed, thus reading in the file which defines 'pname' as a procedure, and the procedure invocation will then proceed just as if 'pname' had been a built-in function in Maple. In other words, one of the side-effects of evaluating 'pname' in this context is that it will be re-defined to have a procedure as its value. Indeed, this method is precisely how the names of Maple's system-defined library functions are initially defined so that the appropriate files will be automatically loaded when needed.

It should be noted that the **readlib** function has option remember and therefore a procedure once loaded will not be re-loaded by subsequent calls to the **readlib** function. This fact can be exploited in the sense that a call to **readlib** provides the functionality: "load if not already loaded". Thus a useful format is to load all necessary functions via **readlibs** at the beginning of a procedure body. Every invocation of such a procedure will execute the **readlib** function calls, but these will cost essentially nothing due to option remember. (See also the description of **readlib** in section 9.1.)

7. OPERATORS

7.1. Operator definition

An *operator* is an abstract data type which describes (at various possible levels: totally, partially, or minimally) an operation to be performed on its arguments. This abstract data type is closely associated with the operations of *application* and *composition,* but will also allow most (or all) of the other algebraic operations.

We will use two main examples to describe operators:

(a) how to represent the first derivative of f(x) at 0, i.e. f'(0) ;

(b) how to represent and to operate with a non-commutative multiplication operator, for example matrix multiplication.

It is important to note that the above examples present three issues to resolve:

(i) a purely representational/syntactic issue: how to input and output these operators;

(ii) a purely functional issue: how to perform all the operations we want performed;

(iii) an integrational issue: how to introduce operators harmoniously into a symbolic algebra system.

In Maple, operators can be:

	Examples
an unassigned name	D
an "ampersand name"	&+, &rank
a procedure	proc(x,y) if y=0 then x^2 else x/y fi end
an "angle bracket expression"	<x^2+sin(x)+1>
any algebraic combination of the above	exp(D) = 1 + <f(x+1)>

Composition is a binary operation (denoted by the symbol @). With this operation, operators form a semigroup with the identity operator (<x>) as identity element.

Application is a binary operation which takes an operator and a list of arguments and applies the rules defined by the operator (if any) on the arguments.

7.2. Syntactic definition

Unassigned names in Maple, since there are no declarations, can represent any data type. As one instance of such they can represent operators.

The ampersand names are just like any other name in Maple, except that they have a special syntactic definition. They are formed by an ampersand followed by any non-null sequence of non-punctuation, non-spacing characters. These can be used as unary or binary operators in the input.

E.g. a &+ b &– c
&det &inv A

An "angle bracket" operator is a shorthand for a single-statement procedure:

< expression | [[parameters]] | [[locals]] >

where [[parameters]] and/or [[locals]] may be omitted. If the parameters are omitted, all the names appearing in the expression which are not system constants are considered to be parameters. (The order of the parameters will be chosen by the system.) E.g.

< x^2 + y^2 >	takes two arguments and computes the sum of their squares
< sin(x) | x >	defines the sin operator (semantically identical to "sin")
< x^2 + b | x >	squares the argument and adds b
< <diff(f(x), x)| x> | f | x >	a possible definition of differentiation operator based on diff of expressions

Repeated composition is denoted by @@. By definition:

a @ a @ ... (n times) ... @ a ≡ a @@ n

The @@ operator has the same precedence as the exponentiation operator and does not associate to the left nor to the right. The symbols @ and @@ were chosen to emphasize the similarities with the ordinary multiplication and exponentiation symbols * and **, and to have a composition symbol as close as possible to the circle operator often used in mathematics to represent composition.

Application uses the "function application" notation; that is, juxtaposition of the operator on the left with a sequence of arguments in parentheses on the right. E.g.

```
f(x)
D(sin)
<x^2 + 2*sin(x)>(Pi)
```

7.3. Semantic definition

Operators, in any of their forms, can be manipulated like any algebraic expressions in Maple. They can be assigned, tested, passed as parameters, involved in algebraic expressions, etc. Additionally we have the following semantic rules.

Testing equality between operators: Parameters and local variables of operators are considered nameless for comparison purposes. Only their number and position count. E.g.

```
<x^2+y | x,y> ≡ <w+z^2 | z,w>
<x^2+y | x,y> ≠ <x^2+y | y,x>
<x> ≡ <y>
```

The semantics of *composition* and *repeated composition* can be illustrated by examples (here a, b, and c represent operators):

```
a @ ( b @ c )   ≡   ( a @ b ) @ c
    a @ <x>     ≡   a
    <x> @ a     ≡   a
     a @@ 0     ≡   <x>
     a @@ 1     ≡   a
   <x> @@ m     ≡   <x>
a@@m @ a@@n     ≡   a @@ (m+n)
( a @@ m ) @@ n ≡   a @@ (m*n)
 a @ (a@@(-1))  ≡   <x>
```

Application of an angle bracket operator or a procedure will follow the standard rules of procedure evaluation. The angle bracket operators are equivalent to single-statement procedures; more precisely:

```
< expr | params | locals > ≡
        proc( params ) local locals; option operator; expr end .
```

For the remaining cases, application of operators will return an unevaluated result. E.g., if f is not defined, f(x) returns f(x). Additional semantics of application can be illustrated by examples:

$$
\begin{aligned}
(a@b)(x) &\equiv a(b(x)) \\
(a+b)(x) &\equiv a(x) + b(x) \\
(a*b)(x) &\equiv a(x)*b(x) \\
(\text{constant})(x) &\equiv \text{constant} \\
\langle x\hat{}2+y \mid x\rangle(5) &\equiv 25+y
\end{aligned}
$$

The function *unapply* is the inverse of application. Application takes an operator and arguments and produces an expression. Unapply takes the expression and arguments and will produce an operator (whenever this is possible). The unapply function implements the lambda-expressions of the lambda calculus. E.g.

$$\text{unapply}(\ \sin(\exp(x))+\cos(x),\ x\ ) \rightarrow \sin @ \exp + \cos$$

and in general:

$$\text{unapply}(\ \text{expr},\ x\ )\ (x) \equiv \text{expr}$$

The following simplifications are handled automatically (from left to right):

$$
\begin{aligned}
\langle a(x) \mid x\rangle &\equiv a \\
\langle \text{constant} \mid x\rangle &\equiv \text{constant} \\
D @ D @ \ldots(\text{n times})\ldots @ D &\equiv D @@ n \\
(D@@n)(f) &\equiv D(D(D(\ldots \text{ n times } \ldots(D(f)..)) \\
(D@C)(f) &\equiv D(C(f)) \\
\sin @ \arcsin &\equiv \langle x\rangle
\end{aligned}
$$

The latter simplification ($\sin @ \arcsin \rightarrow \langle x\rangle$) is not performed automatically, but it is performed by the function simplify.

7.3.1. Application versus composition

It is important to emphasize that function application and function composition are not equivalent. This issue will not normally arise unless we are dealing with operators on operators or more complicated cases.

The problem can be stated in the following terms. Let A and B be operators. To resolve any ambiguity, we cannot avoid analyzing the types of the operators, so let the types of A and B be

$$
\begin{aligned}
&A: \ X \rightarrow W \\
&B: \ Y \rightarrow Z
\end{aligned}
$$

(i.e., A and B being operators, are of type "mapping", mapping a type X into a type W, etc.). Then the following expressions imply certain conditions, as stated:

$$\begin{array}{rll} A\ @@\ n & \text{iff} & X \equiv W \\ A\ @\ B & \text{iff} & Z \equiv X \\ A(B) & \text{iff} & X \equiv (Y \rightarrow Z) \end{array}$$

For example, if R defines our domain of computation for sin and cos then

$$\begin{array}{l} \sin, \cos : R \rightarrow R \\ D : (R \rightarrow R) \rightarrow (R \rightarrow R) \\ \langle x \rangle : X \rightarrow X \quad \text{(for any type X)} \end{array}$$

hence:

$$\begin{array}{rll} \sin\ @\ \cos & \text{is valid} & \\ \sin(a) & \text{is valid} & \text{(if a is of type R)} \\ D(\sin) & \text{is valid} & \\ D\ @\ D & \text{is valid} & \\ \sin(\sin) & \text{is invalid} & \\ D(D) & \text{is invalid} & \end{array}$$

7.4. Partial definition of operators

In Maple, the mathematical knowledge about a given function is not all resident in a single piece of code. For example, Maple knows about the function "sin" in several ways:

$$\begin{array}{c} \sin(\mathrm{Pi}) \rightarrow 0 \\ \mathrm{evalf}(\sin(\ln(2))) \rightarrow 0.6389612763 \\ \mathrm{taylor}(\sin(x), x=0) \rightarrow x - \frac{x^3}{6} + \frac{x^5}{120} + O(x^7) \\ \mathrm{expand}(\sin(x+\mathrm{Pi}/2)) \rightarrow \cos(x) \\ \mathrm{diff}(\sin(x\hat{}2), x) \rightarrow 2\,x\cos(x^2) \\ \mathrm{evalc}(\sin(\ln(-1)))) \rightarrow I\sinh(\mathrm{Pi}) \\ \sin\ @\ \arcsin \rightarrow \langle x \rangle \end{array}$$

The philosophy in Maple is to store this knowledge in a distributed form; in this case, in the procedures sin, evalf/sin, taylor/sin, expand/sin, diff/sin, evalc/sin, and simplify/sin as opposed to "wiring in" all the knowledge in the functions expand, taylor, etc. It follows that we can describe some properties of a function without defining the function itself.

The function **define** is used to define properties of an operator name. It has two basic formats (see section 9.1 for more details).

(1) Defining operator names via known abstract algebraic objects. E.g.:

```
define( group( `&+`, 0, `&-`) ) .
```

(In the define call we want the ampersand names &+ and &– to be

treated as ordinary names; the back quotes are required to prevent them from being treated as operators.)

(2) Defining operator names to have a set of known properties. E.g.:

```
define( a, associative, identity=0, inverse=b );
```

Then:

```
a(x,0)      →  x
a(x,b(x))   →  0
b(0)        →  0
b(b(x))     →  x
a(x,b(a(y,x)))  →  y
```

Define produces the code for evaluation, simplification, expansion, differentiation, taylor expansion, etc. as needed to handle the operations involving the operator. For example, a matrix multiplication operator could be defined as follows:

```
define( `&*`, binary, identity=I, associative, zero=0 ) .
```

This will define an operator with the following properites:

```
A &* I &* B → A &* B
testeq( A &* B = B &* A ) → false
A &* 0 + 3 * I &* (3 * B) → 9 * B
```

7.5. Example: The differentiation operator D

This operator is rather special and, since it is well known, it is worth expanding on it. First, note that there are two useful, distinct, functionalities in symbolic algebra systems:

(i) differentiate an expression with respect to a variable, which we will denote by diff; e.g.

```
diff( expr, x );
diff( x^2, x ); → 2*x
```

(ii) a differentiation operator which can be used in algebraic expressions or can operate over operators, which we will denote by D; e.g.

```
D(sin) → cos
D( <x> ) → 1
```

When the derivative cannot be computed, e.g. diff(f(x),x) for an undefined f, the expressions remain unevaluated. E.g.

```
diff(f(x),x) → diff(f(x),x)
D(f) → D(f)
```

The first derivative of an unknown function f at 0 is expressed as:

```
D(f)(0)
```

and the n^{th} derivative of f at 0 (even if n is undefined) is expressed as:

```
(D @@ n)(f)(0) .
```

For example, a MacLaurin series operator (truncated to 6 terms) could be defined as:

```
ML := < unapply( sum( (D@@i)(f)(0)*x^i/i!, i=0..6 ),x)| f| i,x>;
ML(sin) → <x-x^3/6+x^5/120 | x>
```

Operators are very useful for manipulating differential equations. In mathematics, (D–a)y means y'–ay; in Maple, the equivalent construction is achieved with (D–A)(y) where A := <a*x | x> , a constant-multiplier operator.

The differential equation $(D^2-a^2)y=0$ is expressed in Maple as:

```
( D@@2 – A@@2 ) (y) = 0 .
```

We can factor this particular operator in the form:

```
D@@2 – A@@2 ≡ (D–A) @ (D+A) ≡ (D+A) @ (D–A) .
```

This is not possible in general, but only because D and A commute and D and A are both linear. The original differential equation becomes

```
( (D+A) @ (D–A) ) (y) = 0
```

and since D(0) = 0, (D+A)(y) = 0 and (D–A)(y) = 0 are both solutions.

Consider now the differential equation which is commonly written in mathematics using the product notation: $(D-a)^2y = 0$. In Maple notation, this equation is expressed more precisely as:

```
( (D–A) @@ 2 )(y) = 0
```

or

```
( (D–A) @ (D–A) )(y) = 0 .
```

This equation has a solution when the first application is equal to 0:

```
(D–A)(y) = 0 → y = C1*exp(a*x) .
```

More generally, there is a solution when the first application gives the result

$$(D-A)(y) = C1*\exp(a*x)$$

which will make the second application equal to 0, yielding

$$y = C1*x*\exp(a*x) + C2*\exp(a*x)$$

for arbitrary constants C1 and C2.

8. INTERNAL REPRESENTATION AND MANIPULATION

8.1. Internal Organization

Maple appears to the user as an interactive "calculator". This mode is achieved by immediately executing any statement which is entered at the user level. The main program reads input, calls the parser, and then calls the statement evaluator for each complete statement encountered. The parser accepts the Maple language which has been kept simple enough to have the LALR(1) property. For each production which is successfully reduced, it creates the appropriate data structure. Maple will read an infinite number of statements; its normal conclusion is achieved by the evaluation (not the parsing) of the ⟦quit⟧ statement. Thus it is possible to write a statement like:

if ⟦condition⟧ then quit fi;

which will terminate execution conditionally.

The basic system, or kernel of Maple, is the part of the system which has been written in a systems implementation language and it gets loaded in its entirety when a Maple session is initiated. Most of the mathematical knowledge of the Maple system is coded in the user-level Maple language and resides as library functions in the external Maple library. The Maple library functions are described in chapter 9; the functions which are part of the Maple kernel, called internal functions, will be discussed here.

The internal functions in Maple can be divided into four distinct groups.

(1) Evaluators.
The evaluators are the main functions responsible for evaluation. There are five types of evaluations: statements (handled by *evalstat*); algebraic expressions (*eval*); boolean expressions (*evalbool*); name forming (*evalname*); and floating-point arithmetic (*evalf*). Although the parser calls only *evalstat*, thereafter there are many interactions between the evaluators. For example, the statement

if a>0 then b.i := 3.14/a fi;

is first analyzed by *evalstat* which calls *evalbool* to resolve the if-condition. Once this is completed, say with a true result, *evalstat* is invoked again to do the assignment, for which *evalname* has to be invoked with the left-hand-side and eval with the right-hand expression. Finally, *evalf* will be called to evaluate the result. Normally, the user will not directly invoke any of the evaluators; these are invoked

automatically as needed. In some circumstances, when a non-default type of evaluation is needed, the user can directly call **evalf**, *evalbool* (**evalb**, as called by the user), and *evalname* (**evaln**).

(2) Algebraic functions.
These are functions which are directly identified with functions available at the user level, and are commonly called "basic functions". Some examples are: taking derivatives (**diff**), picking parts of an expression (**op**), dividing polynomials (**divide**), finding coefficients of polynomials (**coeff**), series computation (**taylor**), mapping a function (**map**), substitution of expressions (**subs**, **subsop**), expansion of expressions (**expand**), finding indeterminates (**indets**), etc. Some functions in this group may migrate to the Maple level (Maple library) and vice versa due to tradeoffs between kernel size and efficiency.

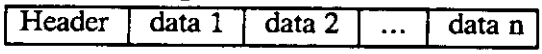
(3) Algebraic service functions.
These functions are algebraic in nature, but serve as subordinates of the functions in the above group. Most frequently these functions cannot be explicitly called by the user. Examples of functions in this group are: the arithmetic (integer, rational, and float) packages, the basic simplifier, printing, the series package, the set-operations package, retrieval of library functions, etc.

(4) General service functions.
Functions in this group are at the lowest hierarchical level; i.e., they may be called by any other function in the system. Their purpose is general, and not necessarily tied to symbolic computation. Some examples are: storage allocation and garbage collection, table manipulation, internal input/output, non-local returns, and various error handlers.

The flow of control within the basic system is not bound to remain at this level. In many cases, where appropriate, a decision is made to call functions written in Maple and residing in the library. For example, most uses of the function **expand** will be handled by the basic system; however, if an expansion of a sum to a large power is required, then the internal **expand** will call the external (Maple library) function `` `expand/bigpow` `` to resolve it. Functions such as **diff**, **evalf**, **taylor**, and **type** make extensive use of this feature. (For example, the basic function **diff** does not know how to differentiate any function; all of its knowledge resides in the Maple library at pathnames `` `diff/[[function name]]` ``.) This is a fundamental feature in the design of Maple as it permits flexibility (changing the library), personal tailoring (defining your own handling functions), readability (the source is in Maple code and available to all users), and it allows the basic system to remain small by unloading non-essential functions to the library.

8.2. Internal Representation of Data Types

The parser and some basic internal functions are responsible for building all of the data structures used internally by Maple. All of the internal data structures have the same general format:



The header field encodes the length of the structure in fifteen bits, the type in six bits, one bit is used to indicate simplification status, and seven bits are used to indicate which generation of storage the object belongs to, the latter being used by the garbage collector. The data items are normally pointers to similar data structures.

Every data structure is created with its own length, and this length will not change during its entire existence. Furthermore, data structures are never changed during execution since it is not predictable how many other data structures are pointing to a given structure. The normal procedure to modify structures is to create a copy and modify the copy, hence returning a new data structure. The only safe modifications are those done by the basic simplifier which produces the same value, albeit simpler. It is the task of the garbage collector to identify unused structures.

In the following figures we will describe the individual structures and the constraints on their data items. We will use the symbolic names of the structures since the actual numerical values used internally are of little interest. The symbol ↑[[xxx]] will indicate a pointer to a structure of type xxx. In particular we will use, whenever possible, the same notation as in the formal syntax (section 4.2).

Logical and

Assignment statement

The [[name]] entry should evaluate to a assignable object, i.e. to one of the following data structures: NAME, FUNCTION or TABLEREF.

Concatenation of a name

CATENATE	↑[[name]]	↑[[expr]]

If the [[name]] entry is one of: NAME, CATENATE, LOCAL, or PARAM, and if the [[expr]] entry evaluates to an integer or to a name, then the result is a name. If [[expr]] is a RANGE then the result is to generate an EXPSEQ. (E.g., a.(1..2) generates a1, a2).

Equation or test for equality

EQUATION	↑[[expr]]	↑[[expr]]

This structure, together with all of the relational operators, has two interpretations depending on the context: as an equation or as a comparison.

Expression sequence

EXPSEQ	↑[[expr]]	↑[[expr]]	...

An EXPSEQ may be of length 1 (no entries); this empty structure is called NULL.

Floating-point number

FLOAT	↑[[integer]]	↑[[integer]]

The floating-point number is interpreted as the first integer times 10 raised to the second integer.

For-while loop statement

FOR	↑[[name]]	↑[[from]]	↑[[by]]	↑[[to]]	↑[[while]]	↑[[statseq]]

or

FOR	↑[[name]]	↑[[in-expr]]	↑[[while]]	↑[[statseq]]

The entries for [[from]], [[by]], [[to]], [[in-expr]], and [[while]] are general expressions which are filled with their default values, if necessary, by the parser. The [[name]] entry follows the same rules as in ASSIGN except that a NULL value indicates its absence. A NULL value in the [[to]] expression indicates that there is no upper limit on the loop.

Function call

FUNCTION	↑[[name]]	↑[[exprseq]]

This structure represents a function invocation (as distinct from a procedure definition which uses the PROC data structure). The [[name]] entry follows the same rules as in ASSIGN, or it may be a PROC definition. The [[exprseq]] contains the list of parameters.

If statement

IF	↑[[cond]]	↑[[statseq]]	↑[[cond]]	↑[[statseq]]	...	↑[[statseq]]

This structure is used to represent if-then, if-then-else, if-then-elif, ... statements. If the length is even, the last entry is an else-statseq. The rest of the entries are interpreted in pairs, 2nd and 3rd, 4th and 5th, etc. Each pair is a condition (an if or an elif condition) and its associated statement sequence.

Not equal or test for inequality

INEQUAT	↑[[expr]]	↑[[expr]]

This structure has two interpretations depending on the context: as an inequation or as a comparison.

Negative integer

INTNEG	integer	integer	...

This data structure is used to represent a negative integer. See the comments below about the representation of integers.

Positive integer

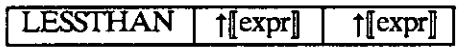
INTPOS	integer	integer	...

This data structure is used to represent a positive integer. Integers are represented in a base that depends on the host machine (10000 for 32-bit machines and 100000 for 36-bit machines). Each entry contains one "digit". A normalized integer contains no additional zeros. The integers are represented in reverse order; i.e., the first entry is the lowest order "digit", the last is the highest order "digit". The base used, BASE, is the largest power of 10 such that $BASE^2$ can be represented in the host-machine integer arithmetic.

Less than or equal

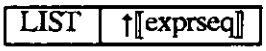
This structure has two interpretations depending on the context: as an inequation or as a comparison. The parser also translates a "greater than or equal" into a structure of this type, interchanging the order of its arguments.

Less than



This structure has two interpretations depending on the context: as an inequation or as a comparison. The parser also translates "greater than" into a structure of this type, interchanging the order of its arguments.

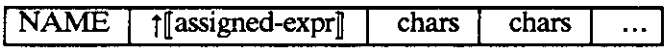
List



Occurrence of a local variable

This entry indicates the usage of the [[integer]]th local variable. This structure is only generated by the simplifier when it processes a function definition. LOCAL entries cannot exist outside functions.

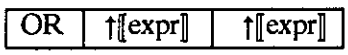
Identifier



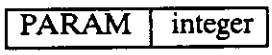
The first entry contains a pointer to the assigned value (if this identifier has been assigned a value) or 0. The chars entries describe the name of the variable. These are null-character terminated, as is the standard in C. The number of characters packed per word is system dependent.

Logical not

Logical or



Occurrence of a parameter variable

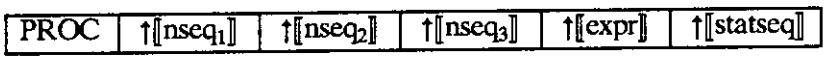


Similar to LOCAL, but indicating the usage of the ⟦integer⟧th parameter of a function.

Power

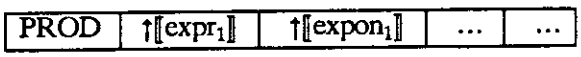
If the second entry is a numeric constant, this structure is changed to a PROD structure by the simplifier.

Procedure definition



The first ⟦nseq⟧ is an EXPSEQ of the names specified for the formal parameters, the second is an EXPSEQ of the names specified for the local variables, and the third is an EXPSEQ of the options specified. The next location, a pointer to an ⟦expr⟧, is for the procedure's remember table (NULL if there is no remember table). The final location is a pointer to the ⟦statseq⟧ which forms the body of the function.

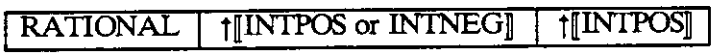
Product/quotient/power



This structure should be interpreted as pairs of factors and their (numeric) exponents. Rational or integer expressions to an integer power are expanded. If there is a rational constant in the product, this constant will be moved to the first entry by the simplifier.

Range

Rational number



The second integer is always positive and different from 0 or 1. The two integers are relatively prime.

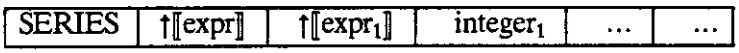
Read statement

The [[expr]] must evaluate to a name (string) which specifies a file.

Save statement

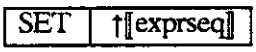
All but the last element in the [[exprseq]] must be the names of expressions to be saved, and the last element (which may be the only element) must evaluate to a name (string) which specifies a file name.

Series



The first expression has the general form x–a where 'x' denotes the 'taylor' variable of the series (i.e., the variable used to do the series expansion) and 'a' denotes the point of expansion. The remaining entries have to be interpreted as pairs of coefficients and exponents. The exponents are integers (not pointers to integers) and appear in increasing order. A coefficient O(1) (function call to the function "O" with parameter 1) is interpreted specially by Maple as an "order" term.

Set



The entries in the [[exprseq]] are sorted in increasing address order. This is an arbitrary but consistent order, necessary for sets.

Statement sequence

End execution

Sum of several terms

This structure should be interpreted as pairs of expressions and their (integer, rational or float) factors. The simplifier lifts as many constant factors as possible from each expression and places them in the [[factor]] entries. An [[expr]] which is a rational constant is multiplied by its factor and then represented with factor 1.

Table

The [[indexing fcn]] will be either a NAME or a PROC. The next location is a null [[expseq]] for general tables. For arrays, it points to an [[exprseq]] of the array bounds specified as a sequence of integer ranges. The [[hashtab]] is not accessible through the use of the **op** function, since it is a hash table storing the table elements.

Table Reference

This data structure represents a table reference, or "indexed name". Its external representation is [[name]] [[[exprseq]]] . The [[name]] entry follows the same rules as in ASSIGN, or it may be a TABLE structure. (The parser will not generate this structure with a TABLE structure for the [[name]] entry, but this may happen internally.) The [[exprseq]] contains the list of indices.

Unevaluated expression

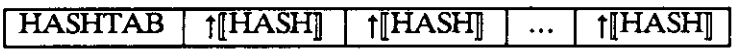
8.3. The Use of Hashing in Maple

Maple's overall performance is in part achieved by the use of table based algorithms for critical functions. Tables are used within the Maple kernel in both evaluation and simplification, as well as less crucial functions. For simplification, Maple keeps a single copy of each expression or subexpression within an entire session. This is achieved by keeping all objects in a table. In procedures, the *remember* option specifies that the result of each computation of the procedure is to be stored in a "remember table" associated with the procedure. Finally, tables are available at the

user level as one of Maple's data types.

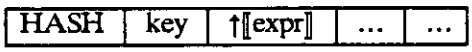
All of the table searching is done by hashing. The algorithm is an implementation of direct chaining in which the hash chains are dynamic vectors instead of linked lists. The two data structures used to implement tables are:

Table entry



Each entry points to a HASH structure or it is 0 if no entry was created. The size of each HASHTAB depends on the type of table and on the size of the system (small systems, large systems). The size of the HASHTAB remains fixed during all its existence. To avoid clustering, the number of HASH entries is chosen to be a prime.

Hash chain vector



Each table element is stored as a pair of consecutive entries in the hash chain vector. The first entry of this pair is the hash key and the second is a pointer to the stored value. In many cases the key is itself a hashing value (two step hashing). A key cannot have the value 0 since this is used as the indicator for the end of a chain. For efficiency, the hash chain vectors are grown a number of entries at a time (proportional to its HASH size) and consequently some of the entries may not be filled.

8.3.1. The Simplification Table

By far, the most important table maintained by the Maple kernel is the simplification table. All simplified expressions and subexpressions are stored in the simplification table. The main purpose of this table is to ensure that simplified expressions have a unique instance in memory. Every expression, which is entered into Maple or generated internally, is checked against the simplification table, and if found, the new expression is discarded and the old one is used. This task is done by the simplifier which recursively simplifies (applies all the basic simplification rules) and checks against the table. Garbage collection deletes the entries in the simplification table which cannot be reached from a global name or from a "live" local variable.

The task of checking for equivalent expressions within thousands of subexpressions would not be feasible if it was not done with the aid of hashing. Every expression is entered in the simplification table using its *signature* as a key. The signature of an expression is a hashing function itself, with one very important attribute: signatures of trivially equivalent expressions are equal. For example, the signatures of the expressions a+b+c and c+a+b are identical; the signatures of a*b and b*a are also identical. If two expressions' signatures disagree then the expressions cannot be equal at the basic level of simplification.

Searching for an expression in the simplification table is done by:

— simplifying recursively all of its components;

— applying the basic simplification rules;

— computing its signature and searching for this signature in the table.

If the signature is found then a full comparison is performed (taking into account that additions and multiplications are commutative, etc.) to verify that it is the same expression. If the expression is found, the one in the table is used and the searched one is discarded. A full comparison of expressions has to be performed only when there is a "collision" of signatures.

Since simplified expressions are guaranteed to have a unique occurrence, it is possible to test for equality of *simplified* expressions using a single pointer comparison. Unique representation of identical expressions is a crucial ingredient of the efficiency of tables, arrays and option remember.

8.3.2. The Name Table

The simplest use of hashing in the Maple kernel is the *name table*. This is a symbol table for all global names. Each key is computed from the name's character string and the entry is a pointer to the data structure for the name. The name table is used to locate global names formed by the lexical scanner or by name concatenation. It is also used by functions that perform operations on all global names. These operations include: (i) marking for garbage collection, (ii) the saving of a Maple session environment in a file, and (iii) the Maple functions **anames** and **unames** which return all assigned global names and all unassigned global names, respectively.

8.3.3. Put Tables

It is possible to store Maple objects in a sequential file using a fast-loading internal format. The pointers in a collection of Maple objects form a general directed acyclic graph (DAG). The process of saving values in a file and later reading the values in from the file (usually in a different session) must preserve this graph, and in particular preserve shared subexpressions. A hash table is temporarily created for each **save** or **read** statement that uses internal format. These tables are known in Maple as *put tables*. The put tables are used to keep track of which subexpressions have already been output to (or input from) the file, and, in general, to perform the mapping from a DAG into a linear (labelled) structure.

8.3.4. Remember Tables

Section 6.5 describes in detail the use of remember tables. Here we describe their internal organization, and which internal functions make use of remember tables. A remember table is a hash table in which the argument(s) to a procedure call are stored as the table index, and the result of the procedure call is stored as the table value. Because a simplified expression in Maple has a unique instance in memory, the address of the arguments can be used as the hash function. Hence searching a remember table is very fast.

There are eight kernel functions that use remember tables: **evalf**, **taylor**, **divide**, **normal**, **expand**, **diff**, **readlib**, and **frontend**. The internal handling of the latter five is straightforward. There are some exceptions with the first three, namely:

— **evalf** and **taylor** need to store some additional, environment, information ('Digits' for **evalf** and 'Order' for **taylor**). Consequently, the entries for these are extended with the precision information. If a result is requested with the same or less precision than what is stored in the table, it is retrieved anyway and "rounded". If a result is produced with more precision than what is stored, it is replaced in the table.

— **evalf** only remembers function calls (this includes named constants); it does not remember the results of arithmetic operations.

— If the division succeeded and the divisor was a non-trivial polynomial, the **divide** function stores the quotient in its remember table. Otherwise nothing is stored in the remember table.

If option remember is specified in conjunction with option system, at garbage collection time the remember table entries which would remain solely because of their presence there (i.e. are not part of any other "live"

expression) are removed. This provides a relatively efficient use of remembering tables, which will not waste storage for expressions which disappeared from the expression space.

8.3.5. Arrays and Tables in the Maple Language

Arrays and tables are provided as data types in the Maple language. An array is a table for which the component indices must be integers lying within specified bounds. Arrays and tables are implemented using Maple's internal hash tables. Because of this, sparse arrays are equally as efficient as dense arrays.

A table object consists of (i) index bounds (for arrays only), (ii) a hash table of components, and (iii) an indexing function. The components of a table T are accessed using a subscript syntax, e.g., T[a,b*cos(x)]. Since a simplified expression is guaranteed to have a unique instance in memory, the address of the simplified index is used as the hash key for a component. If no component exists for a given index, then the indexed expression is returned.

The semantics of indexing into a table are described by its *indexing function*. Aside from the default, general indexing, some indexing functions are provided by the Maple kernel. Other indexing functions are loaded from the library or are supplied by the user. Chapter 5 contains a detailed discussion of arrays and tables, including indexing functions.

8.4. Portability of the Maple System

The Maple system is not tied to one operating system. It has been designed to be portable to many operating systems which support a C compiler in a large-address mode (i.e., at least 22 bit addresses). Maple currently runs under the following systems:

Machine	Operating System
Amdahl mainframes	UTS
Amiga 1000	Amiga DOS
Apollo	Domain IX 9.2
ATT 7300	Unix V.2
Cray 2	Unicos
DEC Microvax	Ultrix 32m
DEC Vax 11	4.2 or 4.3 BSD Unix
DEC Vax 11	VMS 4.4 or 4.5
Gould	Unix
HP 9000/200	HP/UX 2.3
HP 9000/300	HP/UX 5.0
HP 9000/500	HP/UX 5.1
HP Integral	HP/UX 5.0
IBM 43xx	VM/SP CMS Rel 4
IBM RT PC	AIX 1.1
Macintosh Plus, SE	MPW 1.0B2
Masscomp 5xxx	Unix
PCS Cadmus	Munix
Pyramid 96xx	Unix
Siemens PC MX2	Sinix 2.0
Sequent Balance	Dynix
Sequent Symmetry	Dynix
SM-90	Unix
Sun 3	Sun Unix OS
Sun 4	Sun Unix OS

To achieve this level of portability and to have a *single* source code (multiple copies are viewed as a disastrous scenario), we use a general purpose macro-processor called Margay. Margay is a straightforward macro-processor which resembles closely, although it is more powerful than, C's macro-processor. The most important difference is that Margay is written in its own macros and hence it is portable across several systems. The Margay macros reflect machine dependencies, operating system dependencies, size dependencies, and also differences from one C compiler to the next. Another feature of Margay is that it allows functions to have variable-length argument lists.

Once Maple has been ported to a new machine, the user sees a system which is essentially identical to that running on other machines, with a few exceptions such as which character is used as the interrupt character, differences arising from different sizes and speeds, etc. (See the appendices for information on operating system dependencies.)

9. THE MAPLE LIBRARY

Categories of library functions

Maple's library functions fall into four categories: "builtin" functions internal to the Maple system, automatically-loaded library functions, miscellaneous library functions which are not automatically loaded, and packages of functions which are normally loaded via the **with** command.

Functions in the first category are coded in the Maple kernel using the system implementation language, to achieve efficiency and in some cases to achieve functionality that would not be available via user-level Maple code. Functions in the second category are specified by Maple code in the Maple system library, and their names are initially assigned as unevaluated *readlib* defined functions (see section 6.9). Functions in the third category are specified by Maple code and they cannot be used without being explicitly loaded by the user, using the **readlib** function. Packages of functions are specified by Maple code and may be loaded via the **with** library function, as described below.

The distinction between the first two categories will be essentially irrelevant to Maple users. These functions are grouped together and listed alphabetically in section 9.1. For a specific function "f", the user can easily determine which of the two categories it belongs to by entering Maple and displaying the value op('f'). For functions in the first category, the result will be a procedure definition with "options builtin" specified and with an integer for the procedure body, as in:

```
proc () options builtin; 51 end .
```

For functions in the second category, the result will be:

```
readlib('f') .
```

Section 9.2 presents the miscellaneous library functions which are available in the Maple library but which must be explicitly loaded before they can be used. Sections 9.3 through 9.13 present the eleven Maple packages which are currently defined.

Argument evaluation rules

For all functions, all arguments are fully evaluated (from left to right) before being passed to the procedure, with the following three exceptions. The functions *assigned* and *evaln* evaluate their arguments to a name. The function *eval* performs no evaluation of its arguments prior to passing them to the eval procedure, which then performs the specified type of evaluation.

Source codes for library functions

The names of the library functions are not reserved words in Maple. A user may define his own function using the same name as one of the system-supplied functions. The source codes for library functions which are defined by user-level Maple code (i.e., all functions except those in the first category mentioned above) may be accessible for users to read, depending on the particular installation. The location of the library files on a particular host system is determined by the value of the Maple global variable **libname**, which indicates the pathname of the Maple library. Invoking the Maple function call:

```
convert(`` . libname . `/src/f`, hostfile);
```

where `f` is the name of a particular library function, will show the complete pathname of the source file for `f`. (See the **convert** function in section 9.1 and see the appendices for system-dependent information.)

The package concept

The concept of a *package* in Maple permits the user to define an entire collection of new functions with just one Maple command, namely the **with** command. In addition to defining new user-level functions, packages provide two other services. First, they provide the necessary support functions to properly tie these routines into the rest of the system. This includes such facilities as type definitions and rules for expansion of mathematical expressions involving newly-defined functions. Second, they provide convenient short names for each of the routines in the package. In some cases, the effect is to redefine a name which is already used for some other purpose in the Maple session (or a name which is pre-defined by Maple). A warning is issued whenever such a redefinition of a name occurs via the **with** command.

Maple packages load very quickly because, in effect, loading a package requires nothing more than defining the locations of the new functions in the library. The general rule for Maple library functions, including functions defined in packages, is that the code for a function is physically loaded into main memory only when the function is first invoked. For example, execution of the command:

```
with(linalg);
```

causes a table (named `linalg`) to be loaded. The indices of the table are the various function names, and the table entry for each index is the *readlib* definition of the corresponding code for the function. For example, `linalg[transpose]` has the value:

readlib(`linalg/transpose`) .

Furthermore, the **with** command causes the side-effect of assigning to each function name itself (such as the name `transpose`) the corresponding *readlib* definition. Hence, if the package is loaded via `with` then the simple function name `transpose` may be used; otherwise, the long-form notation `linalg[transpose]` would have to be used as the function name.

9.1. Standard library functions

9.1.1. abs – absolute value of real or complex argument

The syntax is abs(e) for an expression e. Automatic simplifications will be applied to this function.

9.1.2. anames – sequence of assigned names

CALLING SEQUENCE : anames( );

- "anames" returns an expression sequence of the names currently assigned values other than their own name.

SEE ALSO : assigned, unames

9.1.3. arc-trigonometric functions

The following function names are used: **arcsin, arccos, arctan, arccsc, arcsec, arccot.** Automatic simplifications will be applied to these functions.

9.1.4. arc-hyperbolic functions

The following function names are used: **arcsinh, arccosh, arctanh, arccsch, arcsech, arccoth.** Automatic simplifications will be applied to these functions.

9.1.5. array – create an array

CALLING SEQUENCE : array(<indexfcn>, <bounds>, <list>)

PARAMETERS :

<indexfcn>	– (optional) an indexing function
<bounds>	– (optional) sequence of ranges
<list>	– (optional) list of initial values

- An array is a specialization of a table, with specified dimensions and each dimension an integer range.
- All parameters to the "array" function are optional and may appear in any order. However, the <bounds> parameter is a sequence of integer ranges which must appear consecutively. If the <bounds> are not specified then they are deduced from the <list> of initial values.
- Only <indexfcn> as op 1, and <bounds> as op 2, are accessible via "op". The <list> of values is only accessible via the array selection operation.

- <indexfcn> can be a procedure or a name specifying how indexing is to be performed; if null then ordinary indexing is implied. Known names are: symmetric, antisymmetric, sparse, sparsymm, diagonal, identity.
- <list> may be a list of equations (cf. tables), or a list of values (one-dimensional), or a nested list of lists (row–by–row).
- Arrays have special evaluation rules (like procedures) so that if the name A has been assigned an array then A evaluates to A; op(A) yields the actual array structure; op(op(A)) yields <indexfcn>, <bounds>.

EXAMPLES :

```
s := array(1..4,1..2);    yields  s := array(1..4,1..2,[])
s[1,2] := x: op(s);       yields  array(1..4,1..2,[(1,2)=x])
A := array( symmetric, 1..2,1..2, [ [x,y], [y,z] ] ):
is equivalent to
A := array( symmetric, 1..2,1..2, [(1,1)=x, (1,2)=y, (2,2)=z] ):
op(1, op(A));             yields  symmetric
op(2, op(A));             yields  1..2,1..2
A[2,1];                   yields  y
```

SEE ALSO : table, indexfcn, symmetric, antisymmetric, sparse, sparsymm, diagonal, identity, copy, selection, op, linalg, evalm

9.1.6. assign – perform assignments

CALLING SEQUENCE : assign(a,b); or assign(a=b); or assign(t)

PARAMETERS :
- a – a name
- b – any expression
- t – a list or set of equations

- For the cases assign(a,b) and assign(a=b) where a is a name, the assignment a := b is made and NULL is returned. Note that the arguments to assign are evaluated. Hence if "a" is already assigned the name "b", then assign(a,2) would assign b the value 2.
- If the argument is a list or set of equations, then assign is applied to each equation in the list or set.
- One application of this function is to apply it to a set of equations returned by the "solve" function when it is desired to make the assignments of the solution values to the variables.

EXAMPLES :

```
assign(a,2); ==> NULL
a; ==> 2
```

```
assign(a=2);  ==>  Error (in assign), invalid arguments
a,b;  ==>  2,2
s := solve( {x+y=1, 2*x+y=3}, {x,y} );  ==>  s := {x=2,y=-1}
assign(s);  ==>  NULL
x,y;  ==>  2,-1
```

9.1.7. assigned – check if a name is assigned

CALLING SEQUENCE : assigned(n);
PARAMETER : n – name, subscripted name, or function call

- The "assigned" function returns true if "n" has a value other than its own name, and returns false otherwise.
- This function is one of the exceptions to the normal evaluation rule for arguments of a function. The argument to "assigned" will only be evaluated as a name (see the "evaln" function) rather than fully evaluated.
- The definition of "assigned" for array/table subscripts is as follows
  assigned(A[i]) = evalb(A[i] <> evaln(A[i]))
  The definition of "assigned" for function calls is as follows
  assigned(f(x)) = evalb(f(x) <> evaln(f(x)))

EXAMPLES :

```
a := table(symmetric);
a[1,2] := x;
assigned(a[1,2]);  ==>  true
assigned(a[2,1]);  ==>  true
assigned(a[1,1]);  ==>  false
assigned(abs(x));  ==>  true
assigned(abs(2));  ==>  false
```

SEE ALSO : anames, unames, evaln

9.1.8. asympt – asymptotic expansion

CALLING SEQUENCE : asympt(<expr>, x, n);
PARAMETERS :

	<expr>	– any function in x
	x	– independent variable
	n	– (optional) truncation order of expansion

- "asympt" computes the asymptotic expansion of expr with respect to the variable x (as x approaches infinity).

- If there is a third argument 'n', it must evaluate to an integer. If there is no third argument the "truncation order" is specified by the value of the global variable, Order (default Order = 6).
- Specifically, "asympt" is defined in terms of the taylor function:
  subs(x=1/x, taylor(subs(x=1/x, expr), x=0, n));
  where the third argument "n" to the taylor function will be omitted if it was omitted in the call to "asympt".
- However, the expression returned will be in sum–of–products form rather than of type series.

9.1.9. bernoulli – Bernoulli numbers and polynomials

CALLING SEQUENCE : bernoulli(n) or bernoulli(n,x);
PARAMETERS : n – a non-negative integer
x – an expression

- The bernoulli function computes the n'th Bernoulli number, or the n'th Bernoulli polynomial, in the expression x. The n'th Bernoulli number B(n) is defined by the exponential generating function
  t/(exp(t)–1) = sum(B(n)/n!*t^n, n=0..infinity) .

EXAMPLES :

```
bernoulli(4);      ==>  –1/30
bernoulli(4,x);    ==>  –1/30 + x^2 – 2*x^3 + x^4
bernoulli(4,1/2);  ==>  7/240
```

SEE ALSO : euler

9.1.10. binomial – binomial coefficients

CALLING SEQUENCE : binomial(n,r);
PARAMETERS : n,r – expressions

- The binomial function computes the binomial coefficients. If the arguments are both positive integers with 0 <= r <= n, then binomial(n,r) = n!/r!/(n–r)! which is the number of distinct ways of choosing r things from n.
- If n and r are integers which do not satisfy 0 <= r <= n, or n and r are rationals or floats, the more general definition binomial(n,r) = GAMMA(n+1)/(GAMMA(r+1)/GAMMA(n–r+1) is used. Otherwise, for symbolic arguments, some simplifications such as binomial(n,1) = n can be made, but, typically, binomial returns unevaluated.

- The function Beta(z,w), defined as int(t^(z−1)*(1−t)^(w−1),t=0..1) can be evaluated using the identity Beta(z,w) = 1/w/binomial(z+w−1,z−1).

EXAMPLES :

```
binomial(4,2);      ==>   6
binomial(2,1/2);    ==>   16/3 1/Pi
binomial(2.1,2);    ==>   1.155
binomial(n,2);      ==>   binomial(n,2)
expand(%);          ==>   1/2*n*(n-1)
```

9.1.11. cat – concatenating expressions

CALLING SEQUENCE : cat(a, b, c, ...);
PARAMETERS : a,b,c, etc. – any expressions

- "cat" is commonly used to concatenate strings together. The arguments are concatenated to form either a name or an object of type `.` .
- Note that the result of cat(a,b,c) is equivalent to doing:
  `` `.`(a,b,c)

SEE ALSO : concat, substring, length

9.1.12. chebyshev – Chebyshev series expansion

CALLING SEQUENCE : chebyshev(<expr>, <eq/nm>, eps);
PARAMETERS :
- <expr> – an algebraic expression or a procedure
- <eq/nm> – an equation x = a..b or a name x
- eps – (optional) a numeric value

- Computes the Chebyshev series expansion of <expr>, with respect to the variable "x" on the interval a..b, valid to accuracy "eps".
- If <eq/nm> evaluates to a name "x" then the equation "x = −1..1" is assumed.
- If the third argument "eps" is present then it specifies the desired accuracy; otherwise, the value used is eps = 10^(−Digits).
- The expression or procedure <expr> must evaluate to a numerical value when x takes on a numerical value. Moreover, it must represent a function which is analytic in a region surrounding the interval a..b.
- The resulting series is expressed in terms of the Chebyshev polynomials T(k,x), k = 0,1,2,... with floating-point series coefficients. Conversion to ordinary polynomial form can be accomplished by simply loading the Chebyshev polynomial definitions via: with(orthopoly,T);

- The series computed is the "infinite" Chebyshev series, truncated by dropping all terms with coefficients smaller than eps times the largest coefficient.

EXAMPLES :

```
Digits := 5:
chebyshev(cos(x), x);          yields .76520*T(0,x)-.22981*T(2,x)
                                  +.0049533*T(4,x)-.000041878*T(6,x)
chebyshev(exp(x), x=0..1, .001); yields 1.7534*T(0,2*x-1)+.85040
                                  *T(1,2*x-1)+.10521*T(2,2*x-1)
                                  +.0087222*T(3,2*x-1)
```

SEE ALSO : orthopoly, taylor

9.1.13. coeff – extract a coefficient of a polynomial

CALLING SEQUENCE : coeff(p,x,n); or coeff(p,x);

PARAMETERS :
- p – a polynomial in x
- x – indeterminate
- n – (optional) degree of x (an integer)

- The coeff function computes the coefficient of x^n in the polynomial p. Note that p must be expanded in the indeterminate x. To expand p in x use collect(p,x) . If the third argument is omitted then it is assumed to be 1.
- The lcoeff function computes the leading coefficient of p in x. The tcoeff function computes the trailing coefficient of p in x. The coeffs function computes all the coefficients of p in x.

EXAMPLES :

```
p := 2*x^2 + 3*y^3 - 5;
coeff( p, x, 2 );   yields  2
coeff( p, x, 1 );   yields  0
coeff( p, x, 0 );   yields  3*y^3 - 5
coeff( p, x^2 );    yields  2
```

SEE ALSO : lcoeff, tcoeff, coeffs

9.1.14. coeffs – extract all coefficients of a multivariate polynomial

CALLING SEQUENCE : coeffs(p, x, 't');

PARAMETERS :
- p multivariate polynomial
- x (optional) indeterminate or list/set of indeterminates
- t (optional) name

- The coeffs function returns an expression sequence of all the coefficients of the polynomial "p" with respect the indeterminate(s) "x".
- If "x" is not specified, coeffs computes the coefficients with respect to all the indeterminates of "p" (see the indets function). If a third argument "t" is specified (call by name), it is assigned an expression sequence of the terms of "p". There is a one to one correspondence between the coefficients and the terms of "p". Note that "p" must be a polynomial in expanded normal form in "x".

EXAMPLES :

```
s := 3*v^2*y^2+2*v*y^3;
coeffs( s );                yields  3, 2
coeffs( s, v, 't' );        yields  3*y^2, 2*y^3
t;                          yields  v^2, v
```

SEE ALSO : coeff, tcoeff, lcoeff, indets

9.1.15. collect – collect terms in a specified indeterminate

CALLING SEQUENCE : collect(a,x);

PARAMETERS : a – any algebraic expression
x – any algebraic expression, or a list or set of such

- The collect function views "a" as a polynomial in "x" and collects all the coefficients with the same power of "x" .
- If "x" is a list or set of variables x1,x2,...,xn, collect first collects terms in x1, then for each coefficient in x1, collects terms in x2 and so on yielding the so called recursive form.
- An optional third argument "distributed" can be specified. If this is the case, collect uses the distributed form. For e1,e2,...,en the coefficients of x1^e1*x2^e2*...*xn^en are collected.

EXAMPLES :

```
collect( (x+1)*(x+2), x );  yields  x^2+3*x+2
collect( y*(sin(x)+1)+sin(x), sin(x) );  yields  (1+y)*sin(x)+y
p := x*y+z*x*y+y*x^2-z*y*x^2+x+z*x:
collect( p, x );  yields  (y-z*y)*x^2+(y+z*y+1+z)*x
collect( p, [x,y] );  yields  (1-z)*y*x^2+((1+z)*y+1+z)*x
collect(p,[x,y],distributed);  yields  (1+z)*x*y+(1-z)*y*x^2+(1+z)*x
```

9.1.16. content and primpart – content / primitive part of a multivariate polynomial

CALLING SEQUENCE : content(a, x, 'pp'); or primpart(a, x, 'co');

PARAMETERS :
- a – multivariate polynomial in x
- x – (optional) name or set or list of names
- pp – (optional) name
- co – (optional) name

- If "a" is a multivariate polynomial with integer coefficients, "content" returns the content of "a" with respect to "x". That is, the greatest common divisor of the coefficients of "a" with respect to the indeterminate(s) "x". The indeterminate(s) "x" can be a name, or a list or set of names.
- The third argument "pp", if present, will be assigned the primitive part, namely "a" divided by the content.
- If the coefficients of "a" in "x" are rational functions then the content computed will be such that the primitive part is a multivariate polynomial over the integers with whose content is 1.
- Similarly, "primpart" returns a/content(a,x). The third argument "co", if present, will be assigned the content. Note: Whereas the "sign" is removed from the content, it is not removed from the primitive part.

EXAMPLES :

```
content( 3 - 3*x, x );                  yields  3
content( 3*x*y + 6*y^2, x );            yields  3 y
content( 3*x*y + 6*y^2, [x,y] );        yields  3
content( -4*x*y + 6*y^2, x, 'pp' );     yields  2 y
pp;                                     yields  - 2 x + 3 y
content( x/a - 1/2, x, 'pp' );          yields  1/2 1/a
pp;                                     yields  2 x - a
primpart( -4*x*y + 6*y^2, x );          yields  - 2 x + 3 y
primpart( x/a - 1/2, x );               yields  2 x - a
```

SEE ALSO : icontent, gcd, coeffs

9.1.17. Content and Primpart – inert content and primpart functions

CALLING SEQUENCE : Content(a, x, 'pp'); or Primpart(a, x, 'co');

PARAMETERS : a – multivariate polynomial in x
x – (optional) name or set or list of names
pp – (optional) name
co – (optional) name

- Content and Primpart are "placeholders" for the content and primitive part of a polynomial over a coefficient domain. They are used in conjunction with mod and evala as described below.
- If "a" is a multivariate polynomial then "content" computes the greatest common divisor of the coefficients of "a" with respect to the indeterminate(s) "x". The indeterminate(s) "x" can be a name, or a list or set of names. The third argument "pp", if present, will be assigned the primitive part, namely "a" divided by the content.
- Similarly, "primpart" returns a/content(a,x). The third argument "co", if present, will be assigned the content. Note: Whereas the "sign" is removed from the content, it is not removed from the primitive part.
- The calls Content(a,x) mod p and Primpart(a,x) mod p compute the content and primitive part respectively modulo the prime integer p. "a" must be a multivariate polynomial over the rationals.
- The calls evala(Content(a,x)) and eval(Primpart(a,x)) do likewise. The polynomial "a" must be a multivariate polynomial with algebraic number coefficients specified by RootOfs.

EXAMPLES :

```
Content(x*(y+4)+y^2+4,x) mod 5;     yields  y+4
Primpart(x*(y+4)+y^2+4,x) mod 5;    yields  x+y+1
```

SEE ALSO : content, primpart, mod, evala, RootOf

9.1.18. convert – convert an expression to a different form

CALLING SEQUENCE : convert(<expr>, <form>, arg3, ...);

PARAMETERS : <expr> – any expression
<form> – a name
arg3, ... – (optional) other arguments

- The convert function is used to convert an expression from one form to another. Some of the conversions are data-type conversions, for example convert([x,y], set); and others are form conversions, for example convert(x^3–3*x^2–7*x+9, horner, x); yields (((x–3)*x–7)*x)+9.
- The types of conversions known are (the second argument <form> must be one of these):

`+`	`*`	array	base	binary	confrac
decimal	degrees	eqnlist	equality	exp	expsincos
factorial	float	fraction	GAMMA	hex	horner
hostfile	lessthan	lessequal	list	listlist	ln
metric	name	octal	parfrac	polar	polynom
radians	radical	rational	ratpoly	RootOf	series
set	sincos	sqrfree			

- For further help do: help(convert,<form>); where <form> is one from the above list.
- A user can make his own conversions known to the convert function by defining a maple procedure in the following way. If the procedure `convert/f` is defined, then the function call convert(a,f,x,y,...); will invoke `convert/f`(a,x,y,...);

EXAMPLES :

```
convert( 9, binary );                 yields  1001
convert( [$(1..10)], `+` );           yields  55
convert( 1.23456, fraction );         yields  3858/3125
convert( (x^2+1)/x, parfrac, x );     yields  x + 1/x
```

SEE ALSO: type

9.1.18.1. convert/`+` and convert/`*` – convert to sum or product

CALLING SEQUENCE : convert(<expression>, `+`)
convert(<expression>, `*`)

- "convert to sum" sums all the ops of the <expression>.
- "convert to product" multiplies all the ops of the <expression>.

EXAMPLES:

```
convert(sin(x)*(y-x),`+`);    yields  sin(x)+y-x
convert(sin(x)+y-x,`*`);      yields  -sin(x)*y*x
convert(f(a,b,c),`+`);        yields  a+b+c
```

9.1.18.2. convert/array – convert to array

CALLING SEQUENCE : convert(<object>, array, <bounds>, <indexfcn>)

PARAMETERS :
- <object> – a table, list or array
- <bounds> – (optional) dimensions of the new array
- <indexfcn> – (optional) indexing function of new array

- "convert/array" converts a table, list, or array to an array. The bounds or index function can be specified.

EXAMPLE:
```
convert([[1,2],[3,4]],array,0..1,-1..0);
  yields  array(0..1,-1..0,[(0,-1)=1,(0,0)=2,(1,-1)=3,(1,0)=4]);
t := table([(1,1)=1,(1,2)=3,(2,2)=4]):
convert(t,array,symmetric);
  yields  array(symmetric,1..2,1..2,[(1,1)=1,(1,2)=3,(2,2)=4]);
convert(",array,3..4,3..4);
  yields  array(3..4,3..4,[(3,3)=1,(3,4)=3,(4,3)=3,(4,4)=4]);
```
SEE ALSO: array

9.1.18.3. convert/base – convert between bases

CALLING SEQUENCE : convert(<number>, base, <beta>)
: convert(<list>, base, <alpha>, <beta>)
PARAMETERS : <alpha> (optional) base of input
<beta> base of output

- In the first case, "convert/base" converts the base 10 number to base <beta>.
- In the second case, it converts the list of digits in base <alpha> to base <beta>. The list of digits is interpreted as the number sum(n[k]*alpha^(k-1), k=1..nops(n)). It is assumed that 0 <= n[k] < alpha and that n is positive.
- The number is returned as a list of digits in base <beta>.

EXAMPLES:
```
convert(17,base,3);          yields  [1,2,2]
convert([1,2,2],base,3,10);  yields  [1,7]
convert(1000,base,60);       yields  [16,40]
```
SEE ALSO: help(convert,binary), help(convert,octal), help(convert,hex), help(convert,decimal)

9.1.18.4. convert/binary – convert to binary form

CALLING SEQUENCE : convert(<number>, binary)

- "convert/binary" converts a decimal number to a binary number.
- The binary number is returned as a base 10 number consisting of the digits 1 and 0 only.

EXAMPLES:

```
convert(123,binary);    yields  1111011
convert(-5,binary);     yields  -101
```

SEE ALSO: help(convert,base), help(convert,decimal), help(convert,hex), help(convert,octal)

9.1.18.5. convert/confrac – convert to continued-fraction form

CALLING SEQUENCE : convert(<expr>, confrac)
convert(<expr>, confrac, maxit)
convert(<expr>, confrac, 'cvgts')
convert(<expr>, confrac, maxit, 'cvgts')
convert(<expr>, confrac, var)
convert(<expr>, confrac, var, order)

- Convert to confrac converts a number, series, rational function, or other algebraic expression to a continued-fraction approximation.
- If <expr> is numeric then maxit (optional) is the maximum number of partial quotients to be computed, and cvgts (optional) will be assigned a list of the convergents. A list of the partial quotients is returned as the function value.
- If <expr> is a series and no additional arguments are specified, a continued-fraction approximation (to the order of the series) is computed. It is equivalent to either an (n,n) or (n,n–1) Pade approximant (depending on the parity of the order). By specifying `subdiagonal` as an optional third argument, the continued-fraction computed will be equivalent to an (n,n) or (n–1,n) Pade approximant.
- If <expr> is a ratpoly (quotient of polynomials) in x, the calling sequence is convert(<expr>, confrac, x). The rational form is converted into its associated continued-fraction form as required for efficient evaluation of numerical subroutines.
- If <expr> is any other algebraic expression, the third argument specifies a variable and (optionally) the fourth argument specifies order. The taylor function is applied to the arguments to obtain a series and then case series applies.
- Otherwise, `convert/confrac` is applied to each component of a non-algebraic structure.

EXAMPLES:

```
convert(2.3,confrac);          yields [2,3,3]
convert(21/13,confrac,cvgts);  yields [1,1,1,1,1,2]
cvgts;                         yields [1,2,3/2,5/3,8/5,21/13]
convert(exp(x),confrac,x);     yields
                               1 +x/(1 +x/(–2 +x/(–3 +x/(2 +x/5)))
r := (1+1/2*x+1/12*x^2) / (1–1/2*x+1/12*x^2):
convert(r,confrac,x);          yields 1+12/(x–6+12/x)
```

SEE ALSO: help(convert,ratpoly)

9.1.18.6. convert/decimal – convert to base 10

CALLING SEQUENCE : convert(<number>, decimal, binary)
convert(<number>, decimal, octal)
convert(<string>, decimal, hex)

- "convert/decimal" converts a binary, octal, or hexadecimal number to a decimal number.
- The binary or octal number is given as an integer. The hexadecimal number is given as a string.

EXAMPLES:

```
convert(101,decimal,binary);   yields 5
convert(23,decimal,octal);     yields 19
convert(`1A`,decimal,hex);     yields 26
```

SEE ALSO: help(convert,binary), help(convert,octal), help(convert,hex), help(convert,base)

9.1.18.7. convert/degrees – convert radians to degrees

CALLING SEQUENCE : convert(<number>, degrees)

- "convert/degrees" converts an expression from radians to degrees by multiplying it by 180/Pi.

EXAMPLE:

```
convert(Pi/2,degrees);   yields 90*degrees
```

SEE ALSO: help(convert,radians)

9.1.18.8. convert/eqnlist – convert to a list of equations

CALLING SEQUENCE : convert(<table>, eqnlist)
convert(<array>, eqnlist)
convert(<list>, eqnlist)

- "convert/eqnlist" will convert a table, array, or list to a list of equations that define it. This will be of the form [index=element,...].

  For a list, the index will be the sequence of op numbers required to get to the element. Any list element which is an equation is incorporated in the result as an index=element equation.

EXAMPLES:

```
A := array([x,y,z]):
convert(A,eqnlist);          yields  [1=x,2=y,3=z]
B := [2=x,1=y,z]:
convert(B,eqnlist);          yields  [1=y,2=x,3=z]
C := array([[1,2],[3,4]]):
convert(C,eqnlist);          yields  [(1,1)=1,(1,2)=2,(2,1)=3,(2,2)=4]
D := [ [4,3=5],(1,2)=7 ]:
convert(D,eqnlist);          yields  [(1,1)=4,(1,3)=5,(1,2)=7]
```

9.1.18.9. convert/equality; convert/lessthan; convert/lessequal

CALLING SEQUENCE : convert(<relation>, equality)
convert(<relation>, lessthan)
convert(<relation>, lessequal)

- "convert/equality" converts a relation to an equality, by replacing the relation with "=". "convert/lessthan" replaces the relation with "<" and "convert/lessequal" replaces the relation with "<=".
- Note that the inequalities ">" and ">=" are automatically converted to "<" and "<=" by Maple.

EXAMPLE:

```
convert(x<y^2,equality);          yields  x=y^2
convert(x<=sin(x),lessthan);      yields  x<sin(x)
convert(x=y,lessequal);           yields  x<=y
```

9.1.18.10. convert/exp – convert trig functions to exponentials

CALLING SEQUENCE : convert(<expr>, exp);

- "convert/exp" converts all trigonometric functions appearing in the expression to their corresponding exponential form.

EXAMPLES:

```
convert(sin(x),exp);                  yields  (exp(I*x)-1/exp(I*x))/2/I
convert(tanh(x),exp);                 yields  (exp(x)^2-1)/(exp(x)^2+1)
convert(exp(x^2)-2*sinh(x^2),exp);    yields  1/exp(x^2)
```

SEE ALSO: help(convert,expsincos), help(convert,sincos), help(convert,ln)

9.1.18.11. convert/expsincos – convert trig functions to exp, sin, cos

CALLING SEQUENCE : convert(<expr>, expsincos);

- "convert/expsin" converts all trigonometric functions appearing in the expression to sin and cos, and converts all hyperbolic trigonometric functions to exponential form.

EXAMPLES:

```
convert(cot(x),expsincos);    yields  cos(x)/sin(x)
convert(sinh(x),expsincos);   yields  1/2*exp(x)-1/2*exp(-x)
```

SEE ALSO: help(convert,exp), help(convert,sincos)

9.1.18.12. convert/factorial – convert GAMMAs and binomials to factorials

CALLING SEQUENCE : convert(<expr>, factorial)
convert(<expr>, factorial, <indets>)

- Convert GAMMAs and binomials in an expression to factorials.
- If an indeterminate or set of indeterminates is specified, then only GAMMAs and binomials involving a specified indeterminate will be converted to the factorial function.

EXAMPLES:

```
convert(GAMMA(x),factorial);     yields  x!/x
convert(binomial(m,3),factorial) yields  1/6*m!/(m-3)!
```

9.1.18.13. convert/float – convert to floating-point

CALLING SEQUENCE : convert(<expression>, float)

- "convert/float" converts an expression to floating-point to precision given by Digits.
- "convert/float" is just a call to evalf.

EXAMPLES:

```
convert(1/2,float);      yields  0.5000000000
convert(Pi,float);       yields  3.141592654
convert(sin(1),float);   yields  0.8414709848
```

SEE ALSO: Digits, evalf

9.1.18.14. convert/GAMMA – convert factorials and binominals to GAMMAs

CALLING SEQUENCE : convert(<expr>, GAMMA)
convert(<expr>, GAMMA, <indets>)

- Convert factorials and binomials in an expression to the GAMMA function.
- If an indeterminate or set of indeterminates is specified, then only factorials and binomials involving a specified indeterminate will be converted to the GAMMA function.

EXAMPLES:

```
convert(x!,GAMMA);                  yields  GAMMA(x+1)
convert(binomial(m,3),GAMMA);       yields
                GAMMA(m+1)/GAMMA(m-2)/6
convert(x!*y!*z!,GAMMA,{x,y});      yields
                GAMMA(x+1)*GAMMA(y+1)*z!
```

9.1.18.15. convert/hex – convert to hexadecimal form

CALLING SEQUENCE : convert(<number>, hex)

- "convert/hex" converts a positive decimal number to a hexadecimal number.
- The hex number is returned as a string consisting of the characters 0 through 9 and A through F.

EXAMPLES:

```
convert(123456,hex);    yields  1E240
convert(100,hex);       yields  64
```

SEE ALSO: help(convert,base), help(convert,binary), help(convert,decimal), help(convert,octal)

9.1.18.16. convert/horner – convert a polynomial to Horner form

CALLING SEQUENCE : convert(<poly>, horner, <var>);
PARAMETERS : <poly> – polynomial
<var> – optional variable or list/set of variables

- Convert to horner forms the polynomial <poly> in the variable <var> in horner or "nested" form. If only two arguments are specified, this is equivalent to convert(<poly>, horner, indets(<poly>)); If the third argument is a list or set of variables, the conversion is applied recursively to the coefficients of <poly> in the first variable. A list allows for control over the order of the conversions.

EXAMPLES:
convert(x^2+3*x+4,horner,x); yields 4+(3+x)*x
poly := y^2*x^2 + 2*y^2*x + 2*y*x^2 + 4*y*x + x^2 + 2*x:
convert(poly,horner,x); yields (2*y^2+4*y+2+(y^2+2*y+1)*x)*x
convert(poly,horner,[x,y]); yields (2+(4+2*y)*y+(1+(2+y)*y)*x)*x

9.1.18.17. convert/hostfile – convert Maple filename to host filename

CALLING SEQUENCE : convert(<filename>, hostfile)

- Convert to hostfile converts the Maple filename to the actual filename used on the host system.
- The filename returned will differ from system to system.

EXAMPLES:
convert(`[bwchar]/subdir/foo.m`,hostfile);
yields [BWCHAR.SUBDIR]FOO.M on VMS
convert(`bwchar/subdir/foo.m`,hostfile);
yields bwchar/subdir/foo.m on Unix
convert(`foo.m`,hostfile);
yields FOO M (BIN RECFM F LRECL 4 on CMS
SEE ALSO: read, save

9.1.18.18. convert/list – convert to a list

CALLING SEQUENCE : convert(<table>, list)
: convert(<vector>, list)
: convert(<expression>, list)
: convert(<operand>, list, list)
: convert(<operand>, list, `=`)

- "convert/list" will take a table or one dimensional array and make a list of the elements. It will make a list of the operands of an expression.
- With the third operand `list`, "convert/list" will make a list of lists. See help(convert,listlist) for more information.

- With the third operand `=`, "convert/list" will make an equation list. See help(convert,eqnlist) for more information.

EXAMPLES:

```
A := array([x,y,z]):
convert(A,list);                    yields  [x,y,z]
convert(r+s-t,list);                yields  [r,s,-t]
B := table([(3)=123,(7)=456]):
convert(B,list);                    yields  [123,456]
```

9.1.18.19. convert/listlist – convert to a list of lists

CALLING SEQUENCE : convert(<array>, listlist)
: convert(<list>, listlist)

- "convert/listlist" will convert an array or list into a list of lists.
- The list of lists formed from an array is equivalent to the list that would be passed to the `array` function to recreate the original array.
- The list of lists is formed from a list by creating an equation list from the original list, using this list to create an array, and then returning the list of lists that forms the array. Basically, that means the list is viewed as equations defining an array, and then "convert/listlist" is applied to that array.

EXAMPLES:

```
A := array([[1,2],[3,4]]):
convert(A,listlist);                yields  [[1,2],[3,4]]
B := [4=12,2=7,1=8,3=6]:
convert(B,listlist);                yields  [8,7,6,12]
C := [ [1=6,2=4],(2,1)=9,(2,2)=6 ]:
convert(C,listlist);                yields  [[6,4],[9,6]]
```

SEE ALSO: array, help(convert,eqnlist)

9.1.18.20. convert/ln – convert arctrig functions to logarithms

CALLING SEQUENCE : convert(<expr>, ln);

- "convert/ln" converts all inverse trigonometric functions appearing in the expression to their corresponding logarithmic form.

EXAMPLES:

```
convert(arcsin(x),ln);      yields  -I*ln((-x**2)**(1/2)+I*x)
convert(arctanh(x),ln);     yields  1/2*ln((1+x)/(1-x))
```

SEE ALSO: help(convert,exp)

9.1.18.21. convert/metric – convert to metric units

CALLING SEQUENCE : convert(<expr>, metric, <imp/US>);
PARAMETERS: <expr> – any expression (dependent on "form")
metric – an unevaluated name
imp/US – unevaluated names

- convert(<expr>, metric); will convert imperial measurements to metric measurements.
- If a third parameter, "imp" or "US", appears the measures specified will be taken to be imperial or U.S., respectively.
- The following units are known to the "convert" function when it is called to convert from a non-metric expression to a metric one.

acre	acres	bu	bushel	bushels	chain
chains	cm	cord	cords	feet	foot
ft	furlong	furlongs	gal	gallon	gallons
Gals	gill	gills	gr	hr	ins
inch	inches	kg	km	Lb	lbs
light_year	light_years	mi	Mile	miles	MPG
MPH	ounce	Ounces	oz	Ozs	pint
pints	pole	poles	pound	pounds	quart
quarts	yard	yards	yd	yds	

EXAMPLES:
convert(4*ft + 3*inches, metric); yields 1.295400000*m
convert(yards, metric); yields .9144*m
convert(5*gallons, metric, imp); yields 22.7298155*lt
convert(5*gallons, metric, US); yields 18.92705892*lt

9.1.18.22. convert/name – convert an expression to a name

CALLING SEQUENCE : convert(<expression>, name)

- "convert/name" converts the expression to a name.

EXAMPLES:
convert(sin(x^2),name); yields sin((x)^(2))
convert(a+b+c*d,name); yields (a)+(b)+((c)*(d))

9.1.18.23. convert/octal – convert to octal form

CALLING SEQUENCE : convert(<number>, octal)

- "convert/octal" converts a positive decimal number to an octal number.
- The octal number is returned as a base 10 number consisting of the digits 0 through 7.

EXAMPLES:

```
convert(12345,octal);    yields  30071
convert(100,octal);      yields  144
```

SEE ALSO: help(convert,base), help(convert,binary), help(convert,decimal), help(convert,hex)

9.1.18.24. convert/parfrac – convert to partial-fraction form

CALLING SEQUENCE : convert(f, parfrac, x, factored);
PARAMETERS : f – rational function
x – main variable name
factored – (optional) true or false

- Convert to parfrac performs a partial fraction decomposition of the rational function "f" in the variable "x".
- The last argument "factored" is optional. If specified, its value is true if it is not necessary to apply normal to f and denom(f) is already in the desired factored form, false otherwise. The default is factored = false.
- This allows for various types of partial fraction decomposition:
  default: maple's factor routine will be applied to the denominator;
  square-free: apply square-free factorization to the denominator before invoking convert/parfrac with factored = true;
  fully factored: factor the denominator into linear factors over its splitting field before invoking convert/parfrac with factored = true.

EXAMPLES :

```
convert(x^2/(x+2),parfrac,x);          yields  x–2 + 4/(x+2)
convert(x/(x^2–3*x+2),parfrac,x);      yields  2/(x–2) – 1/(x–1)
convert(x/(x–b)^2,parfrac,x);          yields  b/(x–b)^2 + 1/(x–b)
poly := x^5–2*x^4–2*x^3+4*x^2+x–2:
f := 36 / convert(poly,sqrfree,x);     yields  f := 36 / (x–2) / (x^2–1)^2
convert(f,parfrac,x,true);             yields
  4/(x–2) – 12*(x+2)/(x^2–1)^2 – 4*(x+2)/(x^2–1)
convert(f,parfrac,x);                  yields
  4/(x–2) – 9/(x–1)^2 – 3/(x+1)^2 – 4/(x+1)
```

9.1.18.25. convert/polar – convert to polar form

CALLING SEQUENCE : convert(<expression>, polar)
PARAMETER : <expression> – a (complex) expression

- "convert/polar" converts an expression to its representation in polar coordinates.
- The expressions is represented as polar(r,theta) where r is the modulus and theta is the argument of the complex value of the expression.

EXAMPLE:

```
convert(3+4*I,polar);   yields   polar(5,arctan(4/3))
convert(3*I,polar);     yields   polar(3,Pi/2);
```

SEE ALSO: evalc

9.1.18.26. convert/polynom – convert a series to a polynomial

CALLING SEQUENCE : convert(<series>, polynom)

- Convert to polynom converts a series to a polynomial.

EXAMPLES:

```
p := taylor(sin(x),x,5);      yields  x-1/6*x^3+O(x^5)
type(p,polynom);              yields  false
s := convert(p,polynom);      yields  x-1/6*x^3
type(s,polynom);              yields  true
```

SEE ALSO: help(convert,series)

9.1.18.27. convert/radians – convert degrees to radians

CALLING SEQUENCE : convert(<number>, radians)

- "convert/radians" converts an expression from degrees to radians by multiplying it by 180/Pi.
- The expression must be of degree 1 in the indeterminate "degrees".

EXAMPLE:

```
convert(90*degrees,radians);   yields   Pi/2
```

SEE ALSO: help(convert,degrees)

9.1.18.28. convert/radical – convert RootOf to radicals and I

CALLING SEQUENCE : convert(<expr>, radical)

- Change occurrences of RootOf(..) to the corresponding radical notation when possible. The "radical notation" for a^(1/2), when sign(a) = –1, will be (sign(a)*a)^(1/2) * I .

- In particular, RootOf(_Z^2+1) is replaced by I.
- In general, if the RootOf argument is a polynomial in _Z with only one term in _Z, say _Z^n, then the RootOf expression is converted to n-th root notation (modulo the introduction of I as noted above).
- For conversion in the opposite direction use convert(expr,RootOf).

EXAMPLES:
convert(RootOf(_Z^2 + 1), radical); yields I
convert(RootOf(_Z^2 – 2), radical); yields 2^(1/2)
convert(RootOf(_Z^2 + _Z + 1), radical); yields RootOf(_Z^2 + _Z + 1)
eval(subs(RootOf=solve, *)); yields
– 1/2 + 1/2*I*3^(1/2), – 1/2 – 1/2*I*3^(1/2)

SEE ALSO : RootOf, help(convert,RootOf), solve

9.1.18.29. convert/rational – convert float to an approximate rational

CALLING SEQUENCE : convert(<float>, rational)
convert(<float>, fraction)
convert(<float>, rational, <digits>)
convert(<float>, fraction, <digits>)

- "convert/rational" converts a floating-point number to an approximate rational number. The meaning of the type "fraction" is identical with "rational" in this context.
- The accuracy of the conversion will depend on the value of the global variable Digits, or the value of <digits> if specified.

EXAMPLES:

convert(0.125,rational);	yields	1/8
convert(2.345,rational);	yields	469/200
convert(evalf(Pi),rational);	yields	104348/33215

SEE ALSO: Digits

9.1.18.30. convert/ratpoly – convert series to a rational polynomial

CALLING SEQUENCE : convert(<series>, ratpoly);
convert(<series>, ratpoly, <numdeg>, <dendeg>);

- Converts a series to a rational polynomial (rational function). If the first argument is a Taylor series then the result is a Pade approximation, and if it is a Chebyshev series then the result is a Chebyshev-Pade approximation.

- The first argument must be either of type 'taylor' (hence a Taylor series) or else a Chebyshev series (represented as a sum–of–products in terms of the basis functions T(k,x) for integers k).
- If the third and fourth arguments appear, they must be integers specifying the desired degrees of numerator and denominator, respectively. (Note: The actual degrees appearing in the approximant may be less than specified if there exists no approximant of the specified degrees.)
- If the third and fourth arguments are not specified then the degrees of numerator and denominator are chosen to be m and n, respectively, such that m+n+1 = order(<series>) and either m=n or m=n+1. (The order of a Chebyshev series is defined to be d+1 where d is the highest-degree term which appears.)
- For the Pade case, two different algorithms are implemented. For the pure univariate case where the coefficients contain no indeterminates, a "fast" algorithm due to Cabay and Choi is used. Otherwise, an algorithm due to Geddes based on fraction-free symmetric Gaussian elimination is used.

EXAMPLES:

```
taylor(exp(x), x);  yields
        1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
convert(", ratpoly);  yields
        (1 + 3/5*x + 3/20*x^2 + 1/60*x^3) / (1 - 2/5*x + 1/20*x^2)
Digits := 5:
chebyshev(cos(x), x);  yields
    .76520*T(0,x)-.22981*T(2,x)+.0049533*T(4,x)-.000041878*T(6,x)
convert(", ratpoly, 2,2);  yields
        (.76025*T(0,x) - .19673*T(2,x)) / (T(0,x) + .043088*T(2,x))
```

9.1.18.31. convert/RootOf – convert radicals and I to RootOf notation

CALLING SEQUENCE : convert(<expr>, RootOf)

- Change all occurrences of the algebraic constant I, and all occurrences of radicals (may be algebraic constants or functions), to RootOf notation.
- In particular, I is replaced by RootOf(_Z^2+1).
- For radical expressions, the base is factored using the "ifactor" routine for integers, or the "factor" routine for more general expressions.
- If the input expression is an unnamed table then the conversion routine is mapped onto the elements of the table.

- To convert RootOf notation back to I and radicals (where possible) use convert(expr,radical).

EXAMPLES :

```
convert( (1 + 2^(1/2)) * I, RootOf );  yields
          (1 + RootOf(_Z^2 - 2)) * RootOf(_Z^2 + 1)
convert( 234543^(1/5), RootOf );  yields
          RootOf(_Z^5 - 3) * RootOf(_Z^5 - 37) * RootOf(_Z^5 - 2113)
convert( (x^2 + 2*x + 1)^(1/2), RootOf );  yields
                                  x + 1
convert( (x^3 - 6*x^2 - 6*x - 7)^(1/2), RootOf );  yields
          RootOf(_Z^2 - x + 7) * RootOf(_Z^2 - x^2 - x - 1)
convert( ", radical );  yields
          (x - 7)^(1/2) * (x^2 + x + 1)^(1/2)
```

SEE ALSO : RootOf, help(convert,radical)

9.1.18.32. convert/series – convert to series

CALLING SEQUENCE : convert(<expression>, series)
convert(<expression>, series, <var>)

PARAMETERS : <expression> – a series or polynomial
<var> – the series variable

- "convert/series" converts a polynomial to a series.
- To convert an arbitrary expression to a series, use the taylor function.
- The parameter <var> is only necessary if <expression> is multivariate.

EXAMPLE:

```
p := x^2+1:
s := convert(p,series);   yields  s := x^2+1;
type(p,series);           yields  false
type(s,series);           yields  true
```

SEE ALSO: taylor, help(convert,polynom)

9.1.18.33. convert/set – convert to a set

CALLING SEQUENCE : convert(<table>, set)
: convert(<array>, set)
: convert(<expression>, set)

- "convert/set" will make a set of the elements of a table or array.

- "convert/set" will make a set of the operands of an expression.

EXAMPLES :

```
A := array(1..2,1..2,[[1,2],[3,4]]):
convert(A,set);                    yields  {1,2,3,4}
convert([a,b,c,d],set);            yields  {a,b,c,d}
convert(x+y-z,set);                yields  {x,y,-z}
```

9.1.18.34. convert/sincos – convert trig functions to sin, cos, sinh, cosh

CALLING SEQUENCE : convert(<expr>, sincos);

- "convert/expsin" converts all trigonometric functions appearing in the expression to sin and cos, and converts all hyperbolic trigonometric functions to sinh and cosh.

EXAMPLES:

```
convert(cot(x),sincos);     yields  cos(x)/sin(x)
convert(tanh(x),sincos);    yields  sinh(x)/cosh(x)
```

SEE ALSO: help(convert,exp), help(convert,expsincos)

9.1.18.35. convert/sqrfree – convert to square-free form

CALLING SEQUENCE : convert(a, sqrfree, x);

PARAMETERS : a – polynomial
x – optional variable name

- Convert to sqrfree performs a square-free factorization of the polynomial "a". It is most often called with the third argument "x" specified. In this case, content(a,x) is first removed (thus making "a" primitive) before doing the square-free factorization. The resulting factorization is of the form:
content(a,x) * p[1]^1 * p[2]^2 * ... * p[k]^k
where gcd(p[i],p[j]) = 1 for 1 <= i < j <= k.
- If called with only two arguments, a "complete" square-free factorization will be done. A variable is first chosen for which the above computation is done. Next, the content is made square-free by recursive application of convert to sqrfree. This is repeated until there are no variables left in the content.

EXAMPLES:
```
convert(x^2+4*x+4,sqrfree,x);    yields   (x+2)^2
poly := y^2*x^3+2*y^2*x^2+y^2*x+2*y*x^3+4*y*x^2+2*y*x
                                          +x^3+2*x^2+x:
convert(poly,sqrfree,x);         yields   (y^2+2*y+1) * (x+1)^2 * x
convert(poly,sqrfree);           yields   (y+1)^2 * (x+1)^2 * x
```
SEE ALSO : content, gcd

9.1.19. copy – create a duplicate array or table

CALLING SEQUENCE : copy(a);
PARAMETER : a – any expression

- The purpose of the "copy" function is to create a duplicate table (or array) which can be altered without changing the original table (or array). If "a" is not a table (or array), "a" is returned.
- This is necessary since the statements s := table(); t := s; leave both names "s" and "t" evaluating to the same table structure. Hence, unlike other Maple data structures, assignments made via one of the names affect the values associated with the other name as well.
- Note that copy creates is not recursive. This means that if "a" is a table of tables, the table data structure for "a" is copied but the table structures for the entries of "a" are not copied.

EXAMPLES :
```
s[1] := x;          yields   s[1] := x
t := s;             yields   t := s
t[1] := y;          yields   t[1] := y
s[1];               yields   y
u := copy(s);       yields   u := table([(1)=y])
u[1] := z;          yields   u[1] := z
s[1];               yields   y
```

9.1.20. define – define characteristics of an operator name

CALLING SEQUENCE : define(aa(oper));
or
define(oper, property1, property2, ...);

PARAMETERS : oper – name of the operator being defined
aa – name of an abstract algebraic object (currently: group and linear)
property.i – name of the property of an operator – or equation defining a value of the operator

- ``define'' defines the evaluation and simplification properties of one operator. It will create the appropriate entries for simplification, evaluation, expand, diff, taylor, testeq, etc. so that the properties of the operator are realized. All these actions are side-effects, ``define'' returns NULL.

  Operators can be used in the infix notation (&–names):

  a &+ b; a &* b &* c; &A x

  or in the functional notation (any name):

  Op(a,b); Op(a); Op(a,b,c)

EXAMPLES :
```
define( linear( `&L` ) );
define( group( G, 0, invG ) );
define( `&+`, associative, commutative, identity=0 );
```
SEE ALSO : operators, neutral, procedures, help(define,property), help(define,linear), help(define,group)

9.1.20.1. define/group – define characteristics of a group operator

CALLING SEQUENCE : define(group(OpName, Identity, Inverse))
PARAMETERS : OpName – is the name of the binary operator defined in the group
Identity – is the identity symbol or value of the group
Inverse – is a unary operator which computes the inverse of the elements in the group

- Sets up a definition of OpName which allows evaluation, simplification and expansion.
- also sets up a definition of the inverse.
  The properties of a group are:
  Closure
  Associativity
  Identity on left and right
  Inverse (unary operator)

EXAMPLES :
```
define( group( `&+`, 0, `&-` ) );
z &+ 0;                          z
&- (&- a);                       a
a &+ (&- a);                     0
```
SEE ALSO : help(define,property)

9.1.20.2. define/linear – define characteristics of a linear operator

CALLING SEQUENCE : define(linear(L));
PARAMETERS : L – is the name of the linear operator being defined

- Sets up a definition of L which allows simplification and evaluation. The properties of a linear operator are:
  L(K1*a + K2*b) = K1*L(a) + K2*L(b)
  L(K3) = K3*L(1)
  for any constants K1, K2 and K3

EXAMPLES :
```
define( linear(L) );
          L(5);   5*L(1)
      L(3*x+z);   3*L(x) + L(z)
         L(Pi);   Pi*L(1)
```
SEE ALSO : help(define,property)

9.1.20.3. define/properties – define an operator by a list of properties

CALLING SEQUENCE : define(oper, property1, property2, ...);
PARAMETERS : oper – name of the operator being defined
property.i – name of the property of an operator
– or equation defining a value of the operator

- ``define" defines the evaluation and simplification properties of one operator. Presently the following properties are processed:
- identity=<expr>; The expression <expr> is considered the left and right identity of the operator.
- binary; Define a binary operator.
- associative; The operator will be treated as an n–ary operator.
- commutative; All the arguments will be ordered to guarantee uniqueness.
- inverse=<operator>; Define the inverse (unary) operator.

- zero=<expr>; Define the "zero" of the operator. If an argument is zero, then the result is zero.
- unary; multiple arguments are disallowed.

EXAMPLES :
define(`&+`, associative, commutative, identity=0);
SEE ALSO : neutral, procedures

9.1.21. degree and ldegree – degree / low degree of a polynomial

CALLING SEQUENCES : degree(a,x); and ldegree(a,x);
PARAMETERS : a – any expression
x – an indeterminate or a list or set of indeterminates

- The "degree" and "ldegree" functions are used to determine the highest degree and lowest degree of the polynomial "a" in the indeterminate "x" which is typically a name but may be any algebraic expression.
- The polynomial "a" may have negative integer exponents in "x". Thus degree and ldegree may return a negative or positive integer. If "a" is not a polynomial in "x" then FAIL is returned.
- If x is a set of indeterminates, the "total" degree/ldegree is computed. If x is a list of indeterminates, then the "vector" degree/ldegree is computed. If x is not specified, this is short for degree(a,indets(a)).

EXAMPLES :

degree(2/x^2+5+7*x^3,x);	yields	3
ldegree(2/x^2+5+7*x^3,x);	yields	–2
degree(x*sin(x),x);	yields	FAIL
degree(x*sin(x),sin(x));	yields	1
degree((x+1)/(x+2),x);	yields	FAIL
degree(x*y^3+x^2,[x,y]);	yields	2
degree(x*y^3+x^2,{x,y});	yields	4
ldegree(x*y^3+x^2,[x,y]);	yields	4

SEE ALSO : lcoeff, tcoeff

9.1.22. diff – partial differentiation

CALLING SEQUENCE : diff(a, x1, x2, ..., xn);
PARAMETERS : a – any algebraic expression
x1, ... – names

- "diff" computes the partial derivative of "a" with respect to x1, x2, ..., xn, respectively.
- Note that where "n" is greater than 1, the call to "diff" is the same as diff called recursively. I.e. diff(f(x),x,y); is equivalent to the call diff(diff(f(x),x),y);
- "diff" has a user interface that will call the user's own differentiation functions. That is, if the procedure `diff/f` is defined then the function call diff(f(x,y,z),y); will invoke `diff/f`(x,y,z,y); to compute the derivative.
- The sequence operator $ is useful for forming higher-order derivatives. E.g., diff(f(x),x$4); is equivalent to diff(f(x),x,x,x,x); and diff(g(x,y),x$2,y$3); is equivalent to diff(g(x,y),x,x,y,y,y);

EXAMPLES :
diff(sin(x),x); yields cos(x)
diff(sin(x),y); yields 0
diff(sin(x),x$3); yields −cos(x)
diff(1/(x+2),x); yields −1/(x+2)^2
SEE ALSO : int, dsolve

9.1.23. divide – exact polynomial division

CALLING SEQUENCE : divide(a, b, 'q');
PARAMETERS : a, b – polynomials with rational number coefficients
q – an unevaluated name

- "divide" checks if the polynomial "b" divides "a" over the rationals. If so true is returned otherwise, false is returned.
- If the division is successful and there is a third argument 'q', then the value of the quotient is assigned to q. In the case of an unsuccessful division the name q will not be affected.

EXAMPLES :
divide(x^3 – y^3, x – y, 'q'); yields true
q; yields x^2 + y*x + y^2
SEE ALSO : quo

9.1.24. Divide – inert divide function

CALLING SEQUENCES : Divide(a,b); or Divide(a,b,'q');
PARAMETERS : a, b – multivariate polynomials
q – unevaluated name

- The "Divide" function is a placeholder for checking the exact polynomial division of "a" by "b" where "a" and "b" are multivariate polynomials. It is used in conjunction with either mod or evala as described below which define the coefficient domain.
- The call Divide(a,b,'q') mod p checks if "b" divides "a" modulo "p", a prime integer. It returns "true" if the division succeeds and assigns the quotient to "q", such that a=b*q; otherwise it returns "false". The polynomials "a" and "b" must have rational coefficients.
- The call evala(Gcd(a,b)) does likewise for "a" and "b" multivariate polynomials with algebraic extensions defined by RootOf's.

EXAMPLES :
```
Divide(x^3+x^2+2*x+3, x+2, 'q') mod 5;   yields true
q;                                       yields x^2+4*x+4
a := x^2+x-2-RootOf(_Z^2-2): b := x-RootOf(_Z^2-2):
evala(Divide(a, b, 'q'));                yields true
q;                                       yields x+1+RootOf(_Z^2-2);
```
SEE ALSO : mod, evala, divide

9.1.25. dsolve – differential equation solver

CALLING SEQUENCE : dsolve(deqn(s), var(s)); or
dsolve(deqn(s), var(s), option);

PARAMETERS :
deqn(s)	– ordinary differential equation in vars, or set of equations and/or initial conditions
var(s)	– variable or set of variables to be solved for
option	– one of "explicit", "laplace", "series"

- dsolve is able to find closed-form solutions to many differential equations. The solution is returned either as an equation in y(x) and x (or whatever variable(s) were specified) or in parametric form [x=f(_T),y(x)=g(_T)] where _T is the parameter.
- "explicit" forces the solution to be returned explicitly in terms of the dependent variable.
- "laplace" causes dsolve to use the Laplace transformation to solve the equation.
- "series" causes dsolve to solve using a series method. The order of the solution can be specified by setting Order.
- To solve a system of differential equations, "laplace" or "series" must be used.

- The initial conditions must be specified at x=0 if the "laplace" or "series" option is being used. Otherwise, the conditions may be initial or boundary conditions specified at any point(s). Derivatives in conditions are specified by additional "p"s added to the function name, e.g. 2nd derivative of y at 0 is ypp(0).

EXAMPLES :

```
dsolve( diff(y(x),x$2) - y(x) = 0, y(x) );          yields
      y(x) = (C*exp(x)^2+C1)/exp(x)
dsolve( {diff(v(t),t)+2*t=0, v(1)=5}, v(t) );       yields
      v(t) = -t^2 + 6
dsolve( diff(y(x),x)-t*y(x)=0, y(x), explicit);     yields
      y(x) = C*exp(x*t)
de1 := diff(y(t),t$2)+5*diff(y(t),t)+6*y(t)=0;
dsolve( {de1,y(0)=0,yp(0)=1}, y(t), laplace);       yields
      y(t) = 2*exp(-5/2*t)*sinh(1/2*t)
sys := diff(y(x),x)=z(x),diff(z(x),x)=y(x):         fcns := {y(x),z(x)}:
dsolve( {sys,y(0)=0,z(0)=1}, fcns, series);         yields
      {y(x) = x+1/6*x^3+1/120*x^5+O(x^6),
          z(x) = 1+1/2*x^2+1/24*x^4+O(x^6)}
```

9.1.26. erf – error function

The error function is defined by

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\mathrm{Pi}}} \int_0^x \exp(-t^2)\, dt \ .$$

Automatic simplifications will be applied to this function.

9.1.27. ERROR – error return from a procedure

CALLING SEQUENCE : ERROR(<expr1>, <expr2>, ...);
PARAMETERS : <expr1>, ... – expression sequence (possibly null)

- A call to the ERROR function causes an immediate return to the point where the current procedure was invoked.
- If the procedure was not invoked within the context of "traperror" then control returns to the top level of the Maple system and the following message is printed: "Error, <in procname>" followed by the evaluated expression sequence specified in the call to ERROR. (Here, procname will be the name with which the procedure was invoked).

- If the procedure was invoked within the context of "traperror" then the value of the global name "lasterror" is assigned the evaluated expression sequence specified in the call to ERROR. In this case, the value of the "traperror" call is also this same expression sequence.

EXAMPLE :

```
f := proc (x) if x<0 then ERROR(`invalid x`, x) else x^(1/2) fi end:
f(−3);                    yields   Error, (in f) invalid x, −3
traperror( f(−3) ):
if " = lasterror then `error occurred` else `no error occurred` fi;
                          yields   error occurred
```

SEE ALSO : procedures, traperror, procname, return, 'lasterror'

9.1.28. euler – Euler numbers and polynomials

CALLING SEQUENCE : euler(n) or euler(n,x)
PARAMETERS : n – a non-negative integer
x – an expression

- The function euler computes the n'th Euler number or, the n'th Euler polynomial in x. The n'th Euler number E(n) is defined by the exponential generating function

  2/(exp(t)+exp(−t)) = sum(E(n)/n!*t^n, n=0..infinity).

EXAMPLES :

```
euler(6);       ==>   −61
euler(4,x);     ==>   x^4 − 2*x^3 + x
euler(4,1/2);   ==>   5/16
```

SEE ALSO : bernoulli

9.1.29. eval – explicit evaluation

CALLING SEQUENCE : eval(x) or eval(x,n)
PARAMETERS : x – an expression
n – an integer

- The normal evaluation rules in Maple are full evaluation for global variables, and one-level evaluation for local variables and parameters. Sometimes the user requires full evaluation or one-level evaluation explicitly. The eval function is used for this purpose.
- eval(x) means fully evaluate the expression x.
- eval(x,1) means evaluate the expression x one level. More generally, eval(x,n) yields n–level evaluation.

- If x := y; y := 1; in a Maple session, what does the variable x evaluate to? In an interactive session where x and y are global variables, x evaluates to 1. We say x is "fully evaluated". For one-level evaluation, we use eval(x,1) which yields y . However, inside a Maple procedure, if x is a local variable or a parameter then the standard mode is one-level evaluation so x evaluates to y . For local variables and parameters, full evaluation is obtained by eval(x) which would yield 1 in this example.
- In normal Maple usage, the user is unlikely to need to use the eval function. The default evaluation rules are usually what is desired.

9.1.30. evala – evaluate in an algebraic number field

CALLING SEQUENCE : evala(expr)
PARAMETER : expr – an expression or an unevaluated function call

- If "expr" is an unevaluated function call (e.g. evala(Gcd(u,v))), then the function is preformed in the smallest algebraic number field possible. See help(function) for more information, where "function" is one of the following:

Content	Divide	Factor	Gcd	Gcdex	Prem
Primpart	Quo	Rem	Resultant	Sprem	

- Otherwise, "expr" is returned unchanged.
- If a dependency is noticed between RootOfs during the computation, then an error occurs, and the dependency is indicated in the error message (this is accessible through the variable "lasterror".)
- An additional argument can be specified for Factor. This is an algebraic number, or a set of algebraic numbers, which are to be included in the field over which the factorization is to be done.

EXAMPLES :
```
evala(Factor(x^2–2), RootOf(_Z^2–2));   yields
  (x–RootOf(_Z^2–2))*(x+RootOf(_Z^2–2))
evala(Quo(x^2 – x + 3, x – RootOf(_Z^2–3), x, 'r'));   yields
  x–1+RootOf(_Z**2–3)
r;   yields
  6–RootOf(_Z**2–3)
```
SEE ALSO : RootOf

9.1.31. evalb – evaluate as a boolean expression

CALLING SEQUENCE : evalb(x);
PARAMETER : x – an expression

- The purpose of the evalb function is to force evaluation of expressions involving relational operators to "true" or "false" if possible; otherwise, it returns an unevaluated expression.
- Normally expressions containing the relational operators =, <>, <, <=, >, >= are treated as algebraic equations or inequalities by Maple. However, when passed as arguments to the "evalb" function (or when appearing in a boolean context in an "if" or "while" statement), they are evaluated to "true" or "false" if possible.
- Note that expressions involving > and >= are converted into equivalent expressions involving < and <=, respectively.

EXAMPLES : If x and y are unassigned then

```
x = x;          yields  x = x
evalb(x=x);     yields  true
evalb(x=y);     yields  false
evalb(x>y);     yields  y – x < 0
```

SEE ALSO : boolean

9.1.32. evalc – evaluate in the complex number field

CALLING SEQUENCE : evalc(<expr>);
PARAMETER : <expr> – an arbitrary expression

- This function is for evaluating and simplifying complex numbers.
- The result of evalc will always be the canonical form
  expr1 + expr2 * I
  involving the symbol I; in particular, (–1)^(1/2) will be converted to the symbol I.
- The result of evalc of a complex series will be a series with each coefficient in the above canonical form.
- Thus if e := evalc(expr); then the "real part" is coeff(e,I,0) and the "imaginary part" is coeff(e,I,1) .
- Or evalc(Re(expr)); and evalc(Im(expr)); return the real and imaginary parts of an expression.
- A complex number may be represented to evalc as polar(r,theta) where r is the modulus and theta is the argument of the number.

EXAMPLES :

```
evalc( (3 + 5*I)*(7 + 4*I) );          yields  1 + 47*I
evalc( Im((3 + 5*I)*(7 + 4*I)) );      yields  47
evalc( (5 – I)/(1 + 2*I) );            yields  3/5 – 11/5*I
evalc( 2^(1+I) );                      yields  2*cos(ln(2)) + 2*sin(ln(2))*I
evalc( exp(I) );                       yields  cos(1) + sin(1)*I
evalc( sin(I) );                       yields  sinh(1)*I
evalc( polar(r,theta) );               yields  r*cos(theta) + r*sin(theta)+I
```

SEE ALSO : polar, help(convert,polar)

9.1.33. evalf – evaluate using floating-point arithmetic

CALLING SEQUENCE : evalf(<expr>, n);

PARAMETERS : <expr> – any expression
n – (optional) integer specifying number of digits

- "evalf" evaluates to floating-point numbers expressions (or subexpressions) involving constants such as Pi, E, gamma, and functions such as exp, ln, sin, arctan, cosh, GAMMA, erf. For the complete list of known constants, see help(ininames); for the complete list of known functions, see help(inifcns);
- The accuracy of the result is determined by the value of the global variable Digits. By default the results will be computed using 10-digit floating-point arithmetic, since the initial value of Digits is 10. A user can change the value of Digits to any positive integer.
- If a second parameter, n, is present the result will be computed using n-digit floating-point arithmetic.
- "evalf" has an interface for evaluating user-defined constants and functions. For example, if a constant "K" is required, then the user must define a procedure called `evalf/constant/K` . Then calls to evalf(K); will invoke `evalf/constant/K`() . For a function example, if the procedure `evalf/f` is defined then the function call evalf(f(2.3, 5.7), 20); will invoke `evalf/f`(2.3, 5.7); and the global variable Digits will be 20 during the evalf call.
- If "evalf" is applied to an unevaluated definite integral then numerical integration will be performed (when possible). To directly invoke numerical integration, use `evalf/int`(f,x=a..b) after first doing readlib(`evalf/int`):

EXAMPLES :

```
evalf( Pi );                          yields  3.141592654
evalf(5/3*exp(-2)*sin(Pi/4), 15);     yields  .159494160850685
evalf(cos(1) + sin(1)*I);             yields  .5403023059 + .8414709848*I
evalf(3/4*x^2+1/3*x-sqrt(2));         yields  .7500000000*x^2 +
                                              .3333333333*x - 1.414213562
int(exp(x^3), x=0..1);                yields  int(exp(x^3), x=0..1)
evalf(%);                             yields  1.341904418
```

SEE ALSO : ininames, inifcns, int

9.1.34. evaln – evaluate to a name

CALLING SEQUENCE : evaln(e);
PARAMETER : e – an expression

- The evaln function is used to "evaluate to a name" or, more generally, to form an assignable object. The name (assignable object) will not itself be evaluated.
- A typical example of its use would be divide(a,b,evaln(t[i])) where i is the index of a for loop.
- If the expression is a simple name (i.e., a string) then the same effect can be achieved by using single quotes on the name.
- Evaln can be used on names, subscripts, function calls, and concatenations.

EXAMPLES :

```
i := 1;                          yields  i := 1
evaln(i);                        yields  i
evaln(a.i);                      yields  a1
evaln(a.(1..3));                 yields  a1, a2, a3
evaln(t[i]);                     yields  t[1]
evaln(f(i));                     yields  f(1)
divide(x^2,x,evaln(t[i]));       yields  true
t[i];                            yields  x
evaln(3);                        yields  Error, illegal use of object as a name
```

SEE ALSO : uneval, assigned, `:=`

9.1.35. exp – the exponential function

The syntax is exp(x) for an expression x. The global constant E is understood to represent exp(1). Automatic simplifications will be applied to this function.

9.1.36. expand – expand an expression

CALLING SEQUENCE : expand(<expr>, e1, e2, . . ., eN)
PARAMETERS : <expr> – any algebraic expression
e1, e2, ..., eN – (optional) expressions

- The primary application of expand is to distribute products over sums. This is done for all polynomials. For quotients of polynomials, only sums in the numerator are expanded – products and powers are left alone.
- "expand" also knows how to expand most of the mathematical functions including sin, cos, tan, sinh, cosh, tanh, det, erf, exp, factorial, GAMMA, ln, Psi, binomial, sum, int, bernoulli, euler, etc.
- The optional arguments e1, e2, ..., en are used to prevent particular sub-expressions in <expr> (e1, e2, ..., en) from being expanded. To prevent all functions from being expanded do: frontend(expand,[<expr>]);
- "expand" has an interface that allows the user to make a function known to expand in the following way. If the procedure `expand/f` is defined, then the call expand(f(x)); will generate the function call `expand/f`(x);

EXAMPLES :

```
expand( (x+1)*(x+2) );          yields  x^2 + 3*x + 2
expand( (x+1)/(x+2) );          yields  x/(x+2) + 1/(x+2)
expand( 1/(x+1)/x );            yields  1/(x+1)/x
expand( sin(x+y) );             yields  sin(x)*cos(y) + cos(x)*sin(y)
expand( cos(2*x) );             yields  2*cos(x)^2-1
expand( exp(a+ln(b));           yields  exp(a)*b
expand( ln(x/(1-x)^2) );        yields  ln(x) - 2*ln(1-x)
expand( (x+1)*(y+z) );          yields  x*y + x*z + y + z
expand( (x+1)*(y+z), x+1 );     yields  (x+1)*y + (x+1)*z
```

SEE ALSO : expandoff, expandon, collect

9.1.37. factor – factor a multivariate polynomial

CALLING SEQUENCE : factor(<expr>);
PARAMETER : <expr> – an algebraic expression

- Factor computes the factorization of a multivariate polynomial with integer or rational coefficients. Note that any integer content is not factored.

- If <expr> is a list, set, equation, or relation, factor is applied recursively to the components of <expr>. If <expr> is a rational function, <expr> is first "normalized" – see normal. The numerator and denominator of the resulting expression are then factored. This provides a "fully-factored form" which can be used to simplify <expr> in the same way the normal function is used. However, it is more expensive than normal.

EXAMPLES :

```
factor( x^2 + 3*x – 4 );    yields  (x+4) * (x–1)
factor( 1 – sin(x)^2 );     yields  – (sin(x)–1) * (sin(x)+1)
factor( {x^3+y^3 = 1} );    yields  {(x+y) * (x^2–x*y+y^2) = 1}
```

9.1.38. Factor – inert factor function

CALLING SEQUENCE : Factor(a)
PARAMETERS : a – multivariate polynomial

- The "Factor" function is a placeholder for representing the factorization of the multivariate polynomial "a" . It is used in conjunction with either mod or evala as described below which define the coefficient domain.
- The call Factor(a) mod p computes the factorization of "a" modulo "p" a prime integer. The multivariate polynomial "a" must have rational coefficients.
- The call evala(Factor(a,e)) computes the factorization of the polynomial "a" over the algebraic number extension "e" which is specified as a RootOf. The multivariate polynomial "a" must have algebraic number coefficients. "e"

EXAMPLE :

```
Factor(x^2+3*x+3) mod 7;    yields  (x+4)*(x+6)
```

SEE ALSO : factor, mod, evala, RootOf

9.1.39. frontend – process a general expression into a rational expression

CALLING SEQUENCE : frontend(p, x, f)
PARAMETERS :
- p – a procedure
- x – a list of arguments to p
- f – (optional) list of two sets:
  first, a set of type names not to be frozen
  second, a set of expressions not to be frozen
  default is [{`+`,`*`},{}]

- The purpose of frontend is to extend the domain of computation for many of the functions in maple. For example, the procedure used by maple's normal function is defined to work over the domain of rational functions. Thus in order to handle more general expressions such as expressions involving sin(x) or sqrt(x) in a reasonable way, frontend is employed to "temporarily freeze" all occurrences of such expressions for unique names. This is always valid. However, it is important to understand that the zero equivalence property of the normal function is only guaranteed if the sub-expressions which are frozen are algebraically independent.
- Frontend is applied to each item in the list x. The order in which the "freezing" occurs is as follows:

  If the argument is atomic, it is not frozen. Integers, names, rationals, and floats are considered to be atomic. If the argument is of type `+` or `*` or `^` and the exponent is an integer, freezing is applied recursively. If the argument has any subexpression in the set of expressions, freezing is applied recursively. If the argument is one of the types in the set of type names, freezing is applied recursively. Otherwise, the expression is substituted for a unique name.
- The procedure f is then evaluated with the "frozen" argument(s). Any frozen names occurring in the result are substituted back for their original subexpressions.

EXAMPLES :
```
a := sin(x)^2 + 2*sin(x) + 1:  b := sin(x) + 1:
gcd(a,b);  yields
  Error, (in gcd) arguments must be polynomials over the rationals
frontend(gcd,[a,b]);  yields  sin(x) + 1
```

9.1.40. fsolve – solve using floating-point arithmetic

CALLING SEQUENCE : fsolve(<eqns>, <vars>, <options>);
PARAMETERS :
- <eqns> – an equation or set of equations
- <vars> – an unknown or set of unknowns
- <options> – (optional) parameters controlling solutions

- The conventions for passing equations and variables, and returning the answers, are the same for fsolve as for solve.
- A range may be passed to fsolve as an option for a single equation. In this case, it will only search for roots in that range.

- For a general equation, fsolve returns a single root. However, for polynomials it will find all real roots, although exceptionally ill-conditioned polynomials may cause fsolve to miss some roots.
- The option fulldigits will prevent fsolve from adjusting Digits for intermediate calculations at high settings of Digits. With this option, fsolve may escape ill-conditioning problems, but the routine slows down.
- When passed maxsols=n as an option, fsolve will only find the n least roots. This option is only meaningful for polynomials, where more than one solution is found.

EXAMPLES :
```
poly := -17*x+10*x^2-105*x^4-23*x^5:
fsolve( poly, x, -1..1 );          yields -.6371813185, 0
fsolve( poly, x, maxsols=3 );      yields -4.536168982, -.6371813185, 0
fsolve( tan(sin(x))=1, x );        yields .9033391108
```
SEE ALSO : solve, dsolve, isolve, msolve, rsolve

9.1.41. GAMMA – the gamma function

The GAMMA function is defined by

$$\mathrm{GAMMA}(x) = \int_0^\infty \exp(-t)\, t^{x-1}\, dt \ .$$

Automatic simplifications will be applied to this function.

9.1.42. gc – garbage collection

CALLING SEQUENCE : gc(n);
PARAMETERS : n – an (optional) integer

- Invocation of this function causes a garbage collection to occur. Garbage collection has the effect of deleting all data (except strings) to which no references are made. It also cleans the remember tables of procedures with option system or option builtin by removing entries that have no other references to them.
- The optional argument n controls the frequency of automatic garbage collections, which will occur after every n words used (approximately). The current value of this frequency is available as status[5], the fifth entry in the 'status' sequence.

- This function returns the value NULL and prints a message showing three values:
  words used=xxxx, alloc=yyyy, time=zzzz
  where xxxx is the total words used, yyyy is the total words actually allocated, and zzzz is the total CPU time used (in seconds) since the start of the session.

SEE ALSO : 'status', words

9.1.43. gcd and lcm – greatest common divisor / least common multiple of polynomials

CALLING SEQUENCES : gcd(a,b,'cofa','cofb');
lcm(a,b,...);

PARAMETERS : a, b – multivariate polynomials over the rationals
cofa,cofb – (optional) unevaluated names

- The gcd function computes the greatest common divisor of two polynomials a and b with rational coefficients.
- The lcm function computes the least common multiple of an arbitrary number of polynomials with rational coefficients.
- The optional third argument cofa is assigned the cofactor a/gcd(a,b).
- The optional fourth argument cofb is assigned the cofactor b/gcd(a,b).

EXAMPLES :
gcd(x^2–y^2,x^3–y^3); yields x – y
lcm(x^2–y^2,x^3–y^3); yields x^4 – x*y^3 – y*x^3 – y^4
gcd(6,8,a,b); yields 2
a; yields 3
b; yields 4

SEE ALSO : gcdex, igcd, ilcm

9.1.44. Gcd – inert gcd function

CALLING SEQUENCES : Gcd(a,b); or Gcd(a,b,'s','t');
PARAMETERS : a, b – multivariate polynomials
s, t – unevaluated names

- The "Gcd" function is a placeholder for representing the greatest common divisor of "a" and "b" where "a" and "b" are multivariate polynomials. It is used in conjunction with either mod or evala as described below which define the coefficient domain.

- The call Gcd(a,b) mod p computes the greatest common divisor of "a" and "b" modulo "p" a prime integer. The polynomials "a" and "b" must have rational coefficients.
- The call evala(Gcd(a,b)) does likewise for "a" and "b" multivariate polynomials with algebraic extensions defined by RootOf's.

EXAMPLE :

```
Gcd(x+2,x+3) mod 7;                          yields  1
Gcd(x^2+3*x+2,x^2+4*x+3,'s','t') mod 11;     yields  x+1
s, t;                                        yields  x+2, x+3
```

SEE ALSO : gcd, mod, evala, RootOf

9.1.45. gcdex – extended Euclidean algorithm for polynomials

CALLING SEQUENCE : gcdex(A,B,x,'s','t'); or gcdex(A,B,C,x,'s','t');
PARAMETERS : A, B, C, x – polynomials in the variable x
s, t – (optional) unevaluated names

- If the number of parameters is less than six, "gcdex" applies the extended Euclidean algorithm to compute polynomials s and t such that s * A + t * B = g (where g is the gcd of A and B as univariate polynomials in x) with degree(s,x) < degree(B,x) and degree(t,x) < degree(A,x). The gcd is returned as the function value.
- In the case of six parameters, "gcdex" solves the polynomial diophantine equation s * A + t * B = C . The input polynomials must satisfy degree(C,x) < degree(A,x) + degree(B,x) and if C is divisible by gcd(A,B) then s and t will be polynomials satisfying degree(s,x) < degree(B,x) and degree(t,x) < degree(A,x). The NULL value is returned as the function value.
- Note that if the input polynomials are multivariate then, in general, s and t will be rational functions in variables other than x.
- In the six-parameter case, if C is not divisible by gcd(A,B) then s and t will be nontrivial rational functions in x.

EXAMPLES:

```
gcdex(x^3-1,x^2-1,x,'s','t');          yields  x-1
s,t;                                   yields  1, -x
gcdex(x^2+a,x^2-1,x^2-a,x,'s','t');
s,t;                                   yields  -(a-1) / (a+1), 2 a / (a+1)
```

SEE ALSO : gcd

9.1.46. Gcdex – inert gcdex function

CALLING SEQUENCES : Gcdex(a,b,x,'s'); or Gcdex(a,b,x,'s','t');
PARAMETERS : a, b – multivariate polynomials
x – main variable
s, t – unevaluated names

- The "Gcdex" function is a placeholder for the extended Euclidean algorithm applied to "a" and "b" which are polynomials in "x" over a field. Gcdex computes "g", the greatest common divisor of "a" and "b", which is a monic polynomial in "x" . Additionally "s" and "t" are assigned (if present) polynomials in "x" such that a*s + b*t = g with degree(s,x) < degree(b,x) and degree(t,x) < degree(a,x). Gcdex is used in conjunction with either mod or evala as described below which define the coefficient domain.
- The call Gcdex(a,b,x,'s','t') mod p performs the computation modulo "p", a prime integer. The multivariate polynomials "a" and "b" must have rational coefficients.
- The call evala(Gcd(a,b,x,'s','t')) does likewise. The multivariate polynomials "a" and "b" must have algebraic number coefficients specified by RootOf's.

EXAMPLE :

```
Gcdex(x^2+x+1,x^2-x+1,x,'s','t') mod 11;   yields  1
s, t;                                      yields  5*x+6, 6*x+6
```

SEE ALSO : gcdex, Gcd, mod, evala, RootOf

9.1.47. genpoly – generate polynomial from integer n by b-adic expansion

CALLING SEQUENCE : genpoly(n, b, x);
PARAMETERS : n – integer or polynomial with integer coefficients
b – integer
x – variable name

- "genpoly" computes the unique polynomial a(x) over Z[x] from the integer n with coefficients less than b/2 in magnitude such that subs(x=b,a(x)) = n.
- This is directly related to b-adic expansion of an integer. If the base–b representation of the integer n is c0 + c1*b + ... + ck*b^k where ci are integers modulo b (using symmetric representation) then the polynomial generated is c0 + c1*x + ... + ck*x^k .

- If n is a polynomial with integer coefficients then each integer coefficient is expanded into a polynomial.

EXAMPLES:
```
genpoly( 11, 5, x );    yields      1 + 2*x
genpoly( 11*y^2-13*y+21, 5, x );            yields
                    (1+2*x)*y^2 + (2+2*x-x^2)*y + (1-x+x^2)
```

9.1.48. has – test for a specified subexpression

CALLING SEQUENCE : has(f, x);
PARAMETERS : f – an expression
x – an expression or list or set of expressions

- If "f" contains the expression "x", then the function "has" will return "true" otherwise it will return "false". The expression "f" contains "x" iff a subexpression of "f" is equal to "x".
- If "x" is a list or set, then "f" contains "x" iff "f" contains at least one item in "x".

EXAMPLES :
```
has( (a+b)^(4/3), a+b );    yields  true
has( (a+b)^(4/3), a );      yields  true
has( a+b+c, a+b );          yields  false
has( a+b+c, {c,d,e} );      yields  true
has( a+b+c, {d,e,f} );      yields  false
```
SEE ALSO : member, op

9.1.49. hastype – test for a specified type

CALLING SEQUENCE : hastype(e, t);
PARAMETERS : e – any expression
t – name of a datatype

- The boolean function hastype returns true if and only if "e" has any subexpressions of the type "t".

EXAMPLES :
```
hastype( (x+y),`+` );                   yields  true
hastype( (x+y)*(1/2*x+y), fraction );   yields  true
hastype( x+7*y, fraction );             yields  false
```
SEE ALSO : has, type

9.1.50. help – descriptions of syntax, datatypes, and functions

CALLING SEQUENCE : help(topic); or help(package, topic);

help(intro);	introduction to Maple
help(index);	list of help categories
help(index, library);	Maple library functions and procedures
help(index, <category>);	list of help files on specific topics
help(<topic>);	explanation of a specific topic
help(<package>, <topic>);	explanation of a topic in <package>
help(`<keyword>`);	explanation of a statement keyword
help(distribution);	for information on how to obtain Maple

Note 1: A Maple keyword (reserved word) cannot be used as an argument to the help function unless it is enclosed in backquotes. For example, "help(quit);" causes a syntax error – use "help(`quit`);" . Note that the string delimiter is backquote (`) not apostrophe (') nor double quote (").
Note 2: A command must end with a semicolon, followed by <RETURN> or <ENTER>, before Maple will execute it and display the result. The semicolon can appear on the next line if you forget to end the command with it, but it must appear. There can be multiple commands on one line, separated by semicolons or colons.
SEE ALSO : keywords, quotes, colons, `quit`

9.1.51. hyperbolic functions

The following function names are used: **sinh, cosh, tanh, csch, sech, coth.** Automatic simplifications will be applied to these functions.

9.1.52. icontent – integer content of a polynomial

CALLING SEQUENCE : icontent(<expr>);
PARAMETER : <expr> – an expanded polynomial with rational coefficients

- "icontent" computes a rational number such that dividing the expanded polynomial by it makes the polynomial primitive over the integers. Thus, for integer coefficients, icontent returns the greatest common divisor of the coefficients.
- For rational coefficients, icontent computes the least common multiple of the denominators of all fractions in addition to the greatest common divisor of the numerators of the coefficients.

EXAMPLES :

```
icontent( 3*x + 12*y );      yields 3
icontent( 7/6*y + 21/4*z );  yields 7/12
```

SEE ALSO : content

9.1.53. ifactor – integer factorization

CALLING SEQUENCE : ifactor(n, method);
PARAMETER : n – any integer
method – (optional) name of base method for factoring

- "ifactor" returns the complete integer factorization of "n".
- The answer is in the form of a product of powers, where the exponent of the powers is an integer and the base of the powers is the null string function, whose argument is an integer.
- The expand function may be applied to cause the factors to be multiplied together again.
- If a second parameter is specified, the named method will be used when the front-end code fails to achieve the factorization. By default, the Morrison-Brillhart algorithm is used as the base method. Currently accepted names are: "squfof", D. Shanks' undocumented square-free factorization; "pollard", J.M. Pollard's rho method; and "easy" which does no further work (the result of the ifactor call will be a product of the factors that were easy to compute, and names c.m indicating m-digit composite numbers that were not factored).
- The pollard base method accepts an additional optional integer: ifactor(n,pollard,k), which increases the efficiency of the method when one of the factors is of the form: k*m+1.

EXAMPLES :

```
ifactor( 61 );          yields  (61)
ifactor( 60 );          yields  (2)^2 (3) (5)
ifactor( -144 );        yields  (-1) (2)^4 (3)^2
expand(%);              yields  -144
ifactor( 60, easy );    yields  (2)^2 (3) (5)
n := 8012940887713968000041:
ifactor( n, easy );     yields  (13) (457) c19
ifactor( n );           yields  (13) (457) (473638939) (2847639359)
```

9.1.54. igcd and ilcm – greatest common divisor / least common multiple of integers

CALLING SEQUENCE : igcd(x1,x2,...);
ilcm(x1,x2,...);
PARAMETERS : x1,x2,... – any integers

- The function "igcd" computes the greatest common divisor of an arbitrary number of integers. The function "ilcm" computes the least common multiple of an arbitrary number of integers.

EXAMPLES :

igcd();	yields	0
ilcm();	yields	1
igcd(–10, 6, –8);	yields	2
ilcm(–10, 6, –8);	yields	120
igcd(a, b);	yields	igcd(a ,b)

SEE ALSO : gcd, lcm

9.1.55. igcdex – extended Euclidean algorithm for integers

CALLING SEQUENCE : igcdex(a, b, 's', 't');
PARAMETERS : a,b – integers
s,t – (optional) names

- igcdex returns g = igcd(a,b), and optionally s and/or t such that g = s a + t b .

EXAMPLES :

igcdex(2,3,'s','t');	yields	1
s; t;	yields	–1 and 1

SEE ALSO : igcd, ilcm, gcdex

9.1.56. indets – find indeterminates of an expression

CALLING SEQUENCE : indets(<expr>); indets(<expr>,<typename>);
PARAMETERS : <expr> – any expression
<typename> – (optional) the name of a type

- "indets" with only one argument returns a set containing all the indeterminates of <expr>.
- <expr> is viewed as a rational expression (i.e. an expression formed by applying only the operations +, –, *, / to some given symbols). Therefore expressions like sin(x), f(x,y), x^(1/2), etc. are treated as indeterminates.

- Expressions of type "constant" such as: sin(1), f(3,5), 2^(1/2), etc., are not considered to be indeterminates in the single-argument case.
- If a second argument <typename> is specified then the value returned is a set containing all subexpressions in <expr> which are of type "typename", including subexpressions which may not have been considered to be indeterminates in the single-argument case.

EXAMPLES :

```
indets( x*y + z/x );                yields {x, y, z}
e := x^(1/2) + exp(x^2) + f(9):
indets(e);                          yields {x, x^(1/2), exp(x^2)}
indets(e,function);                 yields {exp(x^2),f(9)}
```

9.1.57. indices – determine the indices of a table or array

CALLING SEQUENCE : indices(<table>);
PARAMETER : <table> – any table

- "indices" returns an expression sequence containing all the indices (each in the form of a list) of a table.
- The indices will appear in the expression sequence in an apparently arbitrary order which cannot be controlled by the user.

EXAMPLE :

```
t := table([(1,2)=A, (2,1)=B, 9=C]):
indices(t);   yields [1,2], [9], [2,1]
```

9.1.58. int – definite and indefinite integration

CALLING SEQUENCE : int(f, x); int(f, x = a..b);
int(f, x = a..b, cont);

PARAMETERS :

f	–	any algebraic expression in x
x	–	independent variable
a..b	–	interval on which integral is taken
cont	–	(optional) indication that function is continuous

- "int" computes the indefinite or definite integral of "f" with respect to the variable "x".
- Indefinite integration is performed if the second argument is simply "x". Note that no constant of integration appears in the result. Definite integration is performed if the second argument is "x=a..b" where "a" and "b" are the ends of the interval on which the integration is being performed.

- If Maple cannot find a closed form for the integral, the function call itself is returned.
- In the case of a definite integral which returns unevaluated, numerical integration may be invoked by applying "evalf" to the unevaluated integral. To directly invoke numerical integration, use `evalf/int`(f,x=a..b) after first doing readlib(`evalf/int`):
- For definite integration, "iscont" is used to ensure the integrand is continuous before the endpoints are substituted into the indefinite integral. This check can be disabled by calling int with the third option "cont", which will speed up the integration.
- Note that "taylor" may be invoked on an unevaluated integral to compute a series expansion of the integral (when possible).

EXAMPLES :

int(tan(x), x);	yields	−ln(cos(x))
int(ln(x)/x, x);	yields	1/2*ln(x)^2
int(ln(x)/x, x=1..2);	yields	1/2*ln(2)^2
int(exp(−x^2), x);	yields	1/2*Pi^(1/2)*erf(x)
int(exp(x^3), x=0..1);	yields	int(exp(x^3), x=0..1)
evalf(%);	yields	1.341904418
int(1/(x+a)^2,x=0..2,cont);	yields	−1/(1+a)+1/a
int(cos(arctan(x)), x);	yields	int(cos(arctan(x)), x)
taylor(%, x=0);	yields	x−1/6*x^3+3/40*x^5+O(x^7)

SEE ALSO : diff, evalf, taylor, iscont

9.1.59. interp – polynomial interpolation

CALLING SEQUENCE : interp (x, y, v)

PARAMETER : x – list or vector of independent values, x[1],..x[n+1]
y – list or vector of dependent values, y[1],..y[n+1]
v – variable to be used in polynomial

- The function interp computes the polynomial of degree less than or equal to "n" in the variable "v" which interpolates the points (x[1],y[1]), (x[2],y[2]), ..., (x[n+1],y[n+1]) .
- If the same x-value is entered twice, it is an error, whether or not the same y-value is entered; i.e., all independent values must be distinct

SEE ALSO : Interp

9.1.60. Interp – inert interp function

CALLING SEQUENCE : Interp (x, y, v)
PARAMETER : x – list or vector of independent values, x[1],..x[n]
y – list or vector of dependent values, y[1],..y[n]
v – variable to be used in polynomial

- The function "Interp" is a placeholder for polynomial interpolation modulo p. When it is used with "mod" it computes the polynomial of degree at most n–1 that interpolates the points (x[1],y[1]), (x[2],y[2]),..., (x[n],y[n]) mod p .
- If the same x-value mod p is entered twice, it is an error, whether or not the same y-value is entered; i.e., all independent values must be distinct.

EXAMPLES :

Interp([2,5,6], [9,8,3], x);	yields Interp([2,5,6], [9,8,3], x)
Interp([2,5,6], [9,8,3], x) mod 11;	yields 8*x^2 + 6*x + 9

SEE ALSO : interp, mod

9.1.61. irem and iquo – integer remainder / quotient

CALLING SEQUENCE : irem(m,n); or irem(m,n,'q');
iquo(m,n); or iquo(m,n,'r');
PARAMETERS : m, n – any expressions
q, r – names

- If "m" and "n" are both integers the function irem computes the integer remainder of "m" divided by "n". If the third argument is present it will be assigned the quotient. Likewise, iquo computes the integer quotient of "m" divided "n" and if the third argument is present assigns it the remainder.
- Specifically, if "m" and "n" are integers then irem returns "r" such that "m = n*q + r", abs(r) < abs(n) and m*r >= 0 .
- If "m" and "n" are not both integers then "irem" remains unevaluated.

EXAMPLES :

```
irem(23,4,'q');   yields  3
q;                yields  5
iquo(23,4,'r');   yields  5
r;                yields  3
irem(–23,4);      yields  –3
irem(23,–4);      yields  3
irem(–23,–4);     yields  –3
irem(x,3);        yields  irem(x,3)
```

SEE ALSO : rem, quo

9.1.62. isolve – solve equations for integer solutions

CALLING SEQUENCE : isolve(eqns,vars);
PARAMETERS : eqns – a set of equations or a single equation
vars – (optional) set of variables or a variable

- The procedure "isolve" solves the equations in "eqns" over the integers. It solves for all of the indeterminates occurring in the equations.
- The optional second argument "vars" is used to name global variables that have integer values and occur in the solution, and if there is only one argument, then the global names _N1, _N2, etc. are used.
- It returns Warning if there are no integer solutions.

EXAMPLES :

```
isolve(3*x–4*y=7);   yields  {x=4*_N1+5, y=3*_N1+2}
isolve(x^2=3);       yields
                     Warning: unable to find solution over the integers
```

9.1.63. isprime – primality test

CALLING SEQUENCE : isprime(n, iter);
PARAMETER : n – integer
iter – (optional) positive integer (default 5)

- The function isprime is a probabilistic primality testing routine.
- It returns false if "n" is shown to be composite within "iter" tests and returns true otherwise. If isprime returns true, n is "very probably" prime – see Knuth Vol 2, 2nd edition for a reference.

SEE ALSO : nextprime, prevprime, ithprime

9.1.64. isqrt – integer square root

CALLING SEQUENCES : isqrt(x); and iroot(x,n);
PARAMETERS : x, n – integers

- "isqrt" computes an integer approximation to the square root of x. The approximation is exact for perfect squares, and the error is less than 1 otherwise. Note: if $x < 0$, isqrt(x) returns 0.
- "iroot" computes an integer approximation to the n'th root of x. The approximation is exact for perfect powers, and the error is less than 1 otherwise. Note: if $x < 0$, and n is even, iroot(x,n) returns 0.
- The iroot function must first be read in via "readlib(iroot):"

9.1.65. ithprime – determine the i-th prime

CALLING SEQUENCE : ithprime(i);
PARAMETER : i – positive integer

- The function ithprime returns the i'th prime where the first prime is 2.

SEE ALSO : isprime, nextprime, prevprime

9.1.66. lcoeff and tcoeff – leading / trailing coefficient of a multivariate polynomial

CALLING SEQUENCE : lcoeff(p); or lcoeff(p,x); or lcoeff(p,x,'t');
tcoeff(p); or tcoeff(p,x); or tcoeff(p,x,'t');

PARAMETERS :
- p multivariate polynomial
- x (optional) indeterminate or list or set of indeterminates
- t (optional) name

- The functions lcoeff and tcoeff return the leading (trailing) coefficient of "p" with respect the indeterminate(s) "x". If "x" is not specified, lcoeff (tcoeff) computes the leading (trailing) coefficient with respect to all the indeterminates of "p". If a third argument "t" is specified (call by name), it is assigned the leading (trailing) term of "p".
- If "x" is a single indeterminate, and "d" is the degree of "p" in "x", then lcoeff(p,x) = coeff(p,x,d). If "x" is a list or set of indeterminates lcoeff is computes the leading coefficient successively for each indeterminate in "x".
- Note that "p" must be a polynomial in normal form in the indeterminates.

EXAMPLES :

```
s := 3*v^2*w^3*x^4+1;
lcoeff( s );                yields  3
tcoeff( s ); yields  1
lcoeff( s, [v,w], 't' );    yields  3*x^4
t;                          yields  v^2*w^3
```

SEE ALSO : coeff, coeffs, indets, degree, ldegree

9.1.67. length – length of an object

CALLING SEQUENCE : length(n);
PARAMETER : n – any expression

- If n is an integer, the number of decimal digits is returned (where the length of 0 is defined to be 0 and the length of a negative integer is defined to be same as the length of the absolute value of the integer). If n is a string, the number of characters is returned.
- For other objects, the length of each operand of n is computed recursively and added to the number of words used to represent n. This gives a measure of the size of n.

EXAMPLES :

```
length(0); ==> 0
length(11); ==> 2
length(abc); ==> 3
length(x+2*y); ==> 9
```

9.1.68. lexorder – test for lexicographical order

CALLING SEQUENCE : lexorder(<name1>, <name2>);
PARAMETERS : <name1>, <name2> – unevaluated names

- "lexorder" returns "true" if <name1> occurs before <name2> in lexicographical order, or if <name1> is equal to <name2>. Otherwise, it returns false.
- The lexicographical order depends in part upon the ordering of the underlying character set, which is system-dependent.
- For names consisting of ordinary letters, lexicographical order is the standard alphabetical order.

EXAMPLES : (For the ASCII character set)

```
lexorder( a, b );                yields true
lexorder( A, a );                yields true
lexorder( `a`, a );              yields true
lexorder( greatest, great );     yields false
lexorder( `*`, `^` );            yields true
```

9.1.69. limit – calculate function limits

CALLING SEQUENCE : limit(f, <point>, <dir>);

PARAMETERS :

f	– an algebraic expression
<point>	– an equation, x=a, where "x" is a name and "a" is the limit point (possibly infinity or –infinity), or a set of such equations
<dir>	– optional direction: left, right, real, complex

- "limit" attempts to compute the limiting value of f as x approaches a.
- If <dir> is not specified, the limit is the real bidirectional limit, except in the case where the limit point is "infinity" or "–infinity", in which case the limit is from the right to "infinity" and to the left to "–infinity".
- "limit" will assume that any unassigned variables in the expression are real and nonzero (as signum). Thus, limit(a^2*x, x=infinity) will return infinity.
- For further information see help(limit,<topic>) for the topics: multi, dirs, return.
- Based on "A New Algorithm for Computing Symbolic Limits Using Generalized Hierarchical Series" by K.O. Geddes and G.H. Gonnet.

EXAMPLES :

```
limit( sin(x)/x, x=0 );          yields 1
limit( exp(x), x=infinity );     yields infinity
limit( exp(x), x=–infinity );    yields 0
limit( 1/x, x=0, real );         yields undefined
```

9.1.69.1. limit/directional – directional limits

- If <dir> is "left" or "right", the limit is a directional limit. If <dir> is "real", the limit is the bidirectional real limit. If <dir> is "complex", the limit is omni-directional, i.e. from all complex directions to the point a. If <dir> is not specified, the limit is the real bidirectional limit, except in the case where the limit point is "infinity" or "–infinity", in which case the limit is from the left to "infinity" or from the right to "–infinity" respectively.

- If <dir> is "complex", the limit point "infinity" denotes complex infinity, that is, all infinities in the complex plane. If <dir> is "real", the limit point "infinity" denotes both positive and negative infinity, and the limit is done bidirectionally. Otherwise, the limit point "infinity" denotes positive infinity, and "–infinity" denotes negative infinity.

EXAMPLES :

limit(exp(x), x=infinity);	yields infinity
limit(exp(x), x=–infinity);	yields 0
limit(exp(x), x=infinity, real);	yields undefined
limit(1/x, x=0, right);	yields infinity
limit(1/x, x=0, left);	yields –infinity
limit(1/x, x=0);	yields undefined
limit(1/x, x=0, real);	yields undefined
limit(1/x, x=0, complex);	yields infinity
limit(1/x, x=infinity, real);	yields 0
limit(–x, x=infinity, complex);	yields infinity
limit(–x, x=infinity);	yields –infinity

9.1.69.2. limit/multidirectional – multidirectional limits

CALLING SEQUENCE : limit(f, <set–of–points>);
limit(f, <set–of–points>, <dir>);

- "limit" attempts to compute the limiting value of f as in a multi-dimensional space.
- The optional direction <dir> is applied to all points, and consequently all points will be approached from the same direction (for each individual point, that is positive, negative, bidirectional or complex).
- If the limit depends on the direction approached, undefined is returned.
- A limiting point defined as x=x is ignored.

EXAMPLES :

limit((x^2–y^2)/(x^2+y^2), {x=0,y=0});	yields undefined
limit(x+1/y, {x=0,y=infinity});	yields 0
limit(x*y, {x=0,y=infinity});	yields undefined
limit((sin(x^2)–sin(y^2))/(x–y), {x=0,y=0});	yields 0

9.1.69.3. limit/values – values returned by limit

- The meaning of a returned "infinity" depends on <dir>. If <dir> is "complex", then "infinity" denotes complex infinity. Otherwise, the result "infinity" denotes real positive infinity, and "–infinity" denotes real negative infinity.

- If the limit is known to achieve infinitely often any value in a finite real range, then the range is returned.
- If the limit is known to be undefined (assume infinitely often any arbitrary value), or each side of a two-sided limit has a different value, or for a multi-dimensional limit the limiting value depends on the direction at which the limit is approached, then undefined is returned.
- If "limit" is unable to evaluate the limit, then it returns unevaluated. Unless the limit is unevaluated, the limit returned is independent of "x".

EXAMPLES :

	yields
limit(exp(x), x=infinity);	infinity (positive infinity)
limit(1/x, x=0, complex);	infinity (complex infinity)
limit(sin(1/x), x=0);	−1 .. 1
limit(exp(x), x=infinity, real);	undefined
limit(tan(x), x=infinity);	undefined
limit((x^2−y^2)/(x^2+y^2), {x=0,y=0});	undefined
limit(a*x, x=infinity);	signum(a)*infinity

9.1.70. ln – the natural logarithm function

The syntax ln(x) for an expression x represents the natural logarithm of x (logarithm to the base *e*). Automatic simplifications will be applied to this function.

9.1.71. lprint – linear printing of expression

CALLING SEQUENCE : lprint(e1, e2, ...);
PARAMETERS : e1, e2, ... – any expressions

- The function "lprint" returns NULL and prints its arguments in a one-dimensional format.
- The expressions, each separated from the others by three blanks, are printed on a single line.

EXAMPLES :

lprint(3*x^2,x); yields 3*x**2 x

SEE ALSO : print

9.1.72. map – apply a procedure to each operand of an expression

CALLING SEQUENCE : map(<proc>, <expr>, arg2, ..., argN);
PARAMETERS : <proc> – a procedure or a name
<expr> – any expression
argI – (optional) further arguments to <proc>

- The "map" function applies <proc> to the operands of <expr>.
- I.e., the i'th operand of <expr> is replaced by the result of applying <proc> to the i'th operand. This is done for all the operands of <expr>.
- For a table or array, <proc> is applied to each element of the table or array.
- If <proc> takes more than one argument, there must be additional arguments: arg2, arg3 ,..., argn; which are simply passed through as the second, third ,..., n'th arguments to <proc>.

EXAMPLES :

map(f, x + y*z);	yields f(x) + f(y*z)
map(f, y*z);	yields f(y)*f(z)
map(f, {a,b,c});	yields {f(a), f(b), f(c)}
map(proc(x) x^2 end, x + y);	yields x^2 + y^2
map(proc(x,y) x^2+y end, [1,2,3,4], 2);	yields [3,6,11,18]

SEE ALSO : op

9.1.73. max and min – maximum / minimum of numbers

CALLING SEQUENCE : max(x1, x2, ...,);
min(x1, x2, ...,);
PARAMETERS : x1, x2, ... – any expressions

- The functions max and min return the maximum or minimum respectively of one or more arguments.
- Most often the arguments will be of type numeric, that is, integers rationals or floats. However, the functionality is more general allowing for any type of arguments for which an unevaluated function may be returned.

EXAMPLES :

max(3/2, 1.49);	yields	3/2
min(3/2, 1.49);	yields	1.49
max(3/5, evalf(ln(2)), 9/13);	yields	.6931471806
max(x+1, x+2, y);	yields	max(x+2, y)

9.1.74. maxnorm – infinity norm of a polynomial

CALLING SEQUENCE : maxnorm(a);
PARAMETERS : a – an expanded polynomial

- "maxnorm" computes the largest coefficient in absolute value of a polynomial.
- maxnorm understands integer, rational or floating point coefficients.

EXAMPLES:
maxnorm(x–3*y); yields 3

9.1.75. member – test for membership in a set or list

CALLING SEQUENCE : member(x,s,'p');
PARAMETERS : x – any expression
s – a set or list
p – (optional) an unevaluated name

- "member" determines if "x" is a member of the set or list "s". It returns true if so, false otherwise.
- If a third argument "p" is present and member yields true, then the position of the first "x" in "s" will be assigned to "p".

EXAMPLES :

member(y, {x,y,z});	yields	true
member(y, {x*y, y*z});	yields	false
member(x*y, [x*y,w+u,y]);	yields	true
member(w, [x,y,w,u], 'k');	yields	true
k;	yields	3

SEE ALSO : has

9.1.76. mod, modp, and mods – computation over the integers modulo m

CALLING SEQUENCE : e mod m; modp(e,m); mods(e,m); or `mod`(e, m);
PARAMETERS : e – any algebraic expression
m – a nonzero integer

- The mod operator evaluates the expression e in the field of integers modulo m. The default value of `mod` is modp which results in the positive representation for integers modulo m; i.e., all rational coefficients will be reduced to integers in the range [0, abs(m)–1].

- By assigning `mod` := mods; all invocations of mod will then use the symmetric representation; i.e., all rational coefficients will be reduced to integers in the range [–iquo(abs(m)–1,2), iquo(abs(m),2)].
- If "m" does not evaluate to a constant then mod will return an unevaluated function call.
- There is an interface to user-defined mod functions. For example, if the user has defined the procedure `mod/f` then the operation f(x,y) mod 23 will generate the function call `mod/f`(x,y,23).
- The following functions are known to mod:

Content	Divide	Factor	Gcd	Gcdex
Interp	Prem	Primpart	Quo	Rem
Resultant	Roots	Sprem	power	

- For further help do: help(<function>); where <function> is one from the above list.
- To compute i to the power j modulo m, it is undesirable to specify this in the form i^j mod m because the powering will be performed first over the integers (possibly resulting in a very large integer) before reduction modulo m. Rather, the inert operator &^ should be used: i &^ j mod m. In the latter form, the powering will be performed intelligently by the mod operation. "mod" also understands the function name "power" for this same purpose: power(i,j) mod m.
- Other modular arithmetic operations are stated in their natural form: i+j mod m; i–j mod m; i*j mod m; j^(–1) mod m; i/j mod m; where the latter case will perform i*j^(–1) modulo m.
- The left precedence of the mod operator is lower than (less binding strength than) the other arithmetic operators. Its right precedence is immediately higher than +,– and lower than *,/.

EXAMPLES :

```
`mod`;                              yields  modp
12 mod 7;                           yields  5
7 mod m;                            yields  modp(7,m)
`mod` := mods: 12 mod 7;            yields  -2
`mod` := modp: -1 mod 13;           yields  12
3^(-1) mod 7;                       yields  5
17*x*y^11 + 11*x  mod 11;           yields  6*x*y^11
2 &^ 8 mod 100;                     yields  56
`mod` := mods: 4/3 mod 7;           yields  -1
17*x*y^11 + 11*x  mod 11;           yields  -5*x*y^11
a := 9*x^3 + 9*x^2 + 6*x + 6:
b := 3*x^2 + 8*x + 9:
gcd(a,b);                           yields  1
g := Gcd(a,b) mod 11;               yields  g := x + 5
divide(a,g,'q');                    yields  false
Divide(a,g,'q') mod 11;             yields  true
q;                                  yields  -2*x^2 - 3*x - 1
```

SEE ALSO : msolve

9.1.77. msolve – solve equations in Z mod p

CALLING SEQUENCE : msolve(eqns,vars,p); or msolve(eqns,p);

PARAMETERS :
- eqns – a set of equations or a single equation
- vars – (optional) set of names or a name
- p – an integer

- The procedure "msolve" solves the equations in "eqns" over the integers (mod p). It solves for all of the indeterminates occurring in the equations.
- If the solution is indeterminate, and if it is possible, a family of solutions is expressed in terms of the names given in "vars" (or the global names _NN1, _NN2, _NN3 if "vars" is omitted). These names (which should not coincide with the indeterminates) are then allowed to take any integer values.
- It returns NULL if there are no solutions over the integers (mod p).
- Msolve has special code for handling efficiently large systems of equations mod 2.

EXAMPLES :

```
msolve({3*x-4*y=1,7*x+y=2},19);   yields  {x=15, y=11}
msolve(x^2=3,5);                  yields  NULL
msolve(2^i=3,19);                 yields  {i = 13+18*_N}
msolve(8^j=2,x,17);               yields  {j = 3+8*x}
```

SEE ALSO: `mod`

9.1.78. nextprime and prevprime – determine the next largest / smallest prime

CALLING SEQUENCE : nextprime(n); and prevprime(n);
PARAMETER : n – an integer

- "nextprime" returns the smallest prime that is larger than "n". "prevprime" returns the largest prime that is less than "n".

SEE ALSO : isprime, ithprime

9.1.79. normal – normalize a rational expression

CALLING SEQUENCE : normal(<expr>);
normal(<expr>, expanded);
PARAMETER : <expr> – an algebraic expression

- The normal function provides a basic form of simplification. It recognizes those expressions equal to zero which lie in the domain of "rational functions". This includes any expression constructed from sums, products and integer powers of integers and variables. The expression is converted into the what is known as "factored normal form". This is the form numerator / denominator, where the numerator and denominator are relatively prime polynomials with integer coefficients and, any factors in the numerator and denominator are not expanded.
- If <expr> is a list, set, range, series, equation or relation, normal is applied recursively to the components of <expr>. If <expr> contains subexpressions not in the domain of rational functions such as square roots, functions, series etc., normal is first applied recursively to these objects, then they are "frozen" (see the frontend function) to unique names so that the form of <expr> can be simplified. For such cases, normal may not recognize when an expression is equal to zero.
- If normal is called with one argument, the numerator and denominator will be a product of expanded polynomials. If normal is called with the second argument, "expanded", the numerator and denominator will be expanded polynomials.

EXAMPLES :

```
normal( x^2-(x+1)*(x-1)-1 );      yields 0
normal( (x^2-y^2)/(x-y)^3 );      yields (x+y) / (x-y)^2
normal( (f(x)^2-1)/(f(x)-1) );    yields f(x) + 1
normal( {2/x + y/3 = 0} );        yields {1/3 * (6+y*x)/x = 0}
normal( sin(x*(x+1)-x) );         yields sin(x^2)
normal( 1/x+x/(x+1) );            yields (x+1+x^2)/x/(x+1)
normal( 1/x+x/(x+1), expanded );  yields (x+1+x^2)/(x^2+x)
```

SEE ALSO : expand, simplify

9.1.80. numer and denom – numerator / denominator of an expression

CALLING SEQUENCE : numer(e); and denom(e);
PARAMETER : e – any algebraic expression

- Numer and denom are typically called after first using the normal function. Normal is used to put an expression in "normal form" which is the form numerator / denominator where both the numerator and denominator are polynomials. In this case, numer simply picks off the numerator of "e" and denom picks of the denominator of "e". Note that if "e" is in normal form, the numerator and denominator will have integer coefficients.
- If "e" is not in normal form (e contains a subexpression which has one or more terms which are quotients of expressions), it is first converted into a normal form. A common denominator is found so that "e" can be expressed in the form numerator / denominator.

EXAMPLES :

```
numer( 2/3 );                yields 2
denom( 2/3 );                yields 3
numer( x^2-(x-1)*(x+1) );    yields 1
numer( (1+x)/x^(1/2)/y );    yields 1+x
numer( 2/x + y );            yields 2+y*x
numer( x+1/(x+1/x) );        yields x*(x^2+2)
denom( x+1/(x+1/x) );        yields x^2+1
```

SEE ALSO : normal

9.1.81. op and nops – extract operands / number of operands from an expression

CALLING SEQUENCE : op(i,e); op(i..j,e); op(e); and nops(e);
PARAMETERS : i,j – non-negative integers marking positions of operands
e – any expression

- The "op" function extracts the components of an expression. The "nops" function returns the number of components of an expression.
- If the first argument is a nonnegative integer "i" then the result is the i'th operand of "e".
- If the first argument is a range i..j then the result is an expression sequence of the i'th to j'th operands of "e". If there is only one argument: op(e) = op(1..nops(e),e)
- For some data structures, op 0 is defined as a special case. For subscripts and functions, op 0 is the name of the subscript or function and for series, op 0 is the expansion point of the series.

EXAMPLES :

```
u := [1,4,9];     yields  u := [1, 4, 9];
nops(u);          yields  9
op(2,u);          yields  4
op(2..3,y);       yields  4, 9
[op(u),16];       yields  [1, 4, 9, 16]
v := f(x,y,z);
nops(v);          yields  3
op(0,v);          yields  f
op(v);            yields  x, y, z
```

SEE ALSO : type, help(index, datatypes)

9.1.82. order – determine the truncation order of a series

CALLING SEQUENCE : order(<expr>);
PARAMETER : <expr> – an expression of type series

- The function "order" will return the degree of the indeterminate in the O(x) term of a series. If there is no order term in the series then the name "infinity" is returned.
- This function will only work with expressions of type series.

9.1.83. plot – plot functions

CALLING SEQUENCE : plot(f,h,v); or plot(f,h,v,...);
PARAMETERS : f – function(s) to be plotted
h – horizontal range
v – vertical range (optional)

- A typical call to the plot function would be plot(f(x),x=a..b), where "f" is a real function in "x" and a..b specifies the horizontal real range on which "f" is plotted.

- Currently plot provides support for plotting 2 dimensional plots of one or more functions specified as expressions, procedures, parametric functions or, lists of points. See help(plot,function).
- The horizontal and vertical range arguments "h" and "v" define the axis labels and the range over which the function(s) are displayed. They take the form string, or low..hi, or string=low..hi where low and hi are real constants. See help(plot,range).
- Remaining arguments are interpreted as options which are specified as equations of the form option = value. In particular, the style option allows one to plot the points as points only, or interpolate them using by a line or a cubic spline. See help(plot,`options`).
- Assigning the global variable "plotdevice" to one of "char", "tek", "ln03", "postscript" or "regis" results in a character plot, Tektronix 4014 plot, ln03 laser printer plot, postscript-compatible printer or a Regis plot respectively. The default is a character plot. See help(plot,device) and help(plot,setup).

-

EXAMPLES :

```
plot(cos(x),x=0..Pi);              yields a simple plot of cos(x)
plot(tan(x),-Pi..Pi,-5..5);        yields a truncated plot of tan(x)
plot([sin(t),cos(t),t=-Pi..Pi]);   a parametric plot
plot({sin(x),x-x^3/6},0..2);       a multiple plot
plot(sin(x),x=0..infinity);        an infinity plot
l := [0,0,1,1,2,1,2,0,1,-1,0,0]:
plot(l,x=0..2,style=POINT);        plot a list of points
```

SEE ALSO : help(plot,<spec>) where <spec> is one of infinity, polar, parametric, spline, point, line, multiple, ranges, functions, `options`, structure, setup, device, replot.

9.1.83.1. plot/device – explanation of plot devices

USAGE : plotdevice := {one of char, tek, regis, ln03, postscript, vt100}

PARAMETERS :

char	– character plot
tek	– tektronix plot
regis	– regis plot
ln03	– laser printer
postscript	– PostScript-compatible printer
vt100	– vt100 line graphics

- Currently maple supports five plotting devices:
  char – a character plot will be produced. This is the default and will work with any ascii terminal.

regis – will use regis graphics
tek – available on terminals with tektronix emulation.
ln03 – set "plotdevice" to ln03, set "plotoutput" to the name of a temporary file, and run as many plot commands as needed. To print the plots issue the shell command "lpr –Fl –Pprintername filename", where printername is the name of the ln03 printer. This command may be site dependent.
postscript – set "plotdevice" to postscript, set "plotoutput" to the name of a temporary file, and run as many plot commands as needed. To print the plots issue the shell command "lpr –Fl –Pprintername filename", where printername is the name of a PostScript-compatible printer. This command may be site dependent.
vt100 – a plot using the vt100 horizontal line graphics characters will be produced. Especially effective in 132 column mode.

EXAMPLES :
```
plotdevice := regis;
plot(f,h,v,option);
```
SEE ALSO : help(plot,`options`), help(plot,structure), help(plot,setup)

9.1.83.2. plot/function – definition of acceptable plot functions

CALLING SEQUENCE : plot(f,<h>,<v>,options);
PARAMETERS : f – function(s) to be plotted
h – horizontal range
v – vertical range

- Plot in maple accepts three function types: a real function in a single variable expressed as an expression, or a Maple procedure, a parametric function, or a list of points.
- A parametric function has the form [x(t),y(t),range of t]. If the option "coords=polar" is given, the parametric function is interpreted in polar coordinates.
- A list of points has the form [x1,y1,x2,y2,...,xn,yn]. An even number of elements is required.

EXAMPLES :
```
plot(sin(x),–Pi..Pi,–1..1);
plot([sin(t),t^3,7..22]);
plot([5,5,6,6,7,8,9,20]);
```
SEE ALSO : help(plot,parametric), help(plot,`options`), help(plot,polar)

9.1.83.3. plot/infinity – infinity plots

CALLING SEQUENCE : plot(f(x),h,v,<options>);
PARAMETERS : f(x) – an acceptable function
v – horizontal range
v – vertical range (optional)

- If either of the range end points of the horizontal range contain +–infinity, an infinity plot is generated.
- An infinity plot is obtained by transforming –infinity .. infinity to –1 .. 1 by a transformation that approximates arctan. This is a nice way of getting the entire "picture" of f(x) on the display. Such a graph, although distorted near x = –infinity and infinity, contains a lot of information about the features of f(x)

EXAMPLES :

```
plot( sin(x), x=–0..infinity );          infinity plot of sin
plot( {exp,ln}, –infinity..infinity );   infinity plots of exp and ln
```

SEE ALSO : help(plot,ranges), help(plot,`options`), help(plot,function)

9.1.83.4. plot/line – explanation of LINE option

CALLING SEQUENCE : plot(f,h,v,style=LINE)

- There are three styles of plots available on the three types of plot devices. These are: POINT, LINE, SPLINE. The default is SPLINE.
- If the style is LINE, the points computed are connected by straight lines.

EXAMPLES :

```
plot([5,6,7,8,9,10],style=LINE);   a series of points connected
                                   by straight lines
plot(sin(x),–Pi..Pi,style=LINE);   a smooth function represented
                                   by straight lines
```

SEE ALSO : help(plot,`options`), help(plot,structure)

9.1.83.5. plot/multiple – multiple plots

CALLING SEQUENCE : plot({f1,f2,f3,...},h,v,options);
PARAMETERS : f1,f2,f3,... – functions to be plotted
h – horizontal range (optional)
v – vertical range (optional)

- When plot is passed a set of functions it plots all of the passed functions on top of each other.
- In the regis mode three different colours of lines are used. In the tektronix mode eight different line modes are used. In the char mode eight different lines are also used, namely the characters A through H.

EXAMPLES :

```
plot({x-x^3/3,sin(x)},0..1);
plot({x,[x^2,x,-1..1]},0..1);
```

SEE ALSO : help(plot,options), help(plot,functions), help(plot,ranges)

9.1.83.6. plot/option – explanation of plot options

CALLING SEQUENCE : plot(f,h,v,opt1,opt2,...);

PARAMETERS :

f	– some real function
h	– horizontal range (optional)
v	– vertical range (optional)
opt1,opt2,...	– various desired options

- Options to plot are given after the function(s), horizontal range and vertical range as equations of the form option = value . The following options are supported:

coords=polar
: Indicates that a parametric plot is in polar coordinates. The first parameter is the radius and the second parameter is the angle.

numpoints=n
: Specifies the minimum number of points to be generated (default 49 for an infinity plot, 25 otherwise) Note: plot employs an adaptive plotting scheme which automatically does more work where the function values do not lie close to a straight line. Hence plot will often generate more than the minimum number of points.

resolution=n
: Sets the horizontal display resolution of the device in pixels (default 200). The value of n is used to determine when the adaptive plotting scheme terminates. A higher value will result in more function evaluations for non-smooth functions.

style=s
: The interpolation style must be one of (default is SPLINE):
POINT – plot points only
LINE – linear interpolation
SPLINE – cubic spline interpolation

title=t
The title for the plot. t must be a character string. The default is no title.

EXAMPLES :
plot(sin(x),–2*Pi..2*Pi,title=`Sine Graph`); does what you would think
SEE ALSO : help(plot,structure), help(plot,polar),

9.1.83.7. plot/parametric – parametric plots

CALLING SEQUENCE : plot([x(t),y(t),range of t],<h>,<v>,options);
PARAMETERS : [x,y,range] – parametric specifications
h – horizontal range
v – vertical range

- A parametric plot is specified by list of three items, the first two are real functions of a parameter, the third is a range over which the parameter ranges.
- Used in conjunction with the option coords=polar produces polar plots.

EXAMPLES :
plot([sin(t),cos(t),0..Pi]); a parametric plot in t
plot([x^2,x,0..3*Pi],–8..8,–10..10,coords=polar); a cropped polar plot
plot([(t^2–1)/(t^2+1),2*t/(t^2+1), t=–infinity..infinity]);
# the latter is the parametric form for a unit circle
SEE ALSO : help(plot,`options`), help(plot,polar)

9.1.83.8. plot/point – explanation of POINT option

CALLING SEQUENCE : plot(f,h,v,style=POINT);

- There are three styles of plots available on the three types of plot devices. These are: POINT, LINE, SPLINE. The default is SPLINE.
- If the style is POINT, plot simply plots the calculated or given points.

EXAMPLES :
plot([2,3,4,7,2,9],style=POINT); plots three points
plot(sin(x),–Pi..Pi,style=POINT); plots the calculated points
SEE ALSO : help(plot,structure), help(plot,`options`);

9.1.83.9. plot/polar – polar coordinate plots

CALLING SEQUENCE : plot([r(t),theta(t),range of t],<h>,<v>,coords=polar);

PARAMETERS : r(t) – the distance from the origin as a function of t
theta(t) – the angle of rotation in terms of t
h – the horizontal range
v – the vertical range

- If the option "coords=polar" is specified, parametric functions will be interpreted in polar coordinates. The "radius" is the first parameter and the "angle" is the second parameter.

EXAMPLES :
plot([1,t,0..2*Pi],coords=polar); yields a circle
SEE ALSO : help(plot,parametric)

9.1.83.10. plot/range – explanation of plot ranges

CALLING SEQUENCES : plot(f,x=low..hi,y=low..hi,options);
PARAMETERS : f – function(s)
x,y – x and y are the axis labels (optional)
low..hi – a real range (optional)

- The horizontal and vertical range specifications define the axis labels and the range over which the function(s) are displayed.
- If no range is specified, –10..10 is used for the horizontal range. If one range is specified, this is taken as the horizontal range. If the vertical range is unspecified, a suitable range will be deduced from the values computed.
- The low and hi elements of an axis range must be real constants. This includes numbers, +–infinity and constants such as Pi, exp(4).
- If either + or – infinity is included in an axis range specification then an infinity plot is computed.

EXAMPLES :
plot(tan,–Pi..Pi,–5..5); view of tan from –5..5
plot(sin(x),`x axis`=–Pi..Pi,`y axis`);
SEE ALSO : help(plot,`options`), help(plot,infinity)

9.1.83.11. plot/replot – redo a plot

CALLING SEQUENCE : replot(p, h, v, ...);
PARAMETERS : p – a plot data structure
h – horizontal range (optional)
v – vertical range (optional)
... – optional arguments (see below)

- This function allows one to change the parameters of a plot without the plot being recalculated. This allows one to zoom in on a plot or change the title, labels, or style. The options to replot are the same as the options to plot. See help(plot,`options`);
- Note that since replot doesn't recompute the function, it won't provide more data in a new plot than was in the original plot. So changing the scale won't show more of the function or increase any detail. This requires computing the plot again using plot. Also, replot can't switch between infinity and bounded plots.
- Note: If a vertical range is specified, the horizontal range must be specified too, since the first range is assumed to be the horizontal range.
- This function must first be loaded via: readlib(replot):

-

EXAMPLES :

```
readlib(replot):
plot(tan(x),0..Pi);                  yields a plot of tan(x)
replot(*,x=0..Pi,y=-5..5);           yields a clipped plot of tan(x)
replot(*,x=0..Pi/2,title=tan);       yields a plot of tan(x) on 0..Pi/2
p := plot(sin(x),0..2*Pi):           p contains a plot of sin(x)
q := replot(p,x=0..Pi,y=-1..1):      q contains a plot with labelled axes
q;                                   yields the plot of sin(x)
```

SEE ALSO : plot

9.1.83.12. plot/setup – entering graphics modes

- The Maple variables "preplot" and "postplot" are used in defining the strings that a terminal needs to enter and leave graphics mode. These variables should be assigned a list of positive integers in the range 0..127, representing the ASCII codes of the characters to be sent, and negative integers in the range -1..-127 representing delays in seconds. Preplot and postplot are limited to 80 elements. For example, to put a VT240 terminal into Tektronix emulation one would assign

```
preplot := [27,91,63,51,56,104];
```

which corresponds to the characters,

```
ESC [ ? 3 8 h
```

To put a KD500G into Tektronix mode, one would use

```
preplot := [27,49];
```

which corresponds to the characters,

```
ESC 1
```

The corresponding assignments to get the terminal back out of graphics

mode after plotting has completed would be,

```
postplot := [27,91,63,51,56,108];
```

and

```
postplot := [27,50];
```

respectively.
At this point, "preplot" and "postplot" sequences are needed only when the "plotdevice" variable is set to "tek" and Tektronix emulation mode is being used, since various terminals use different sequences to enter and leave Tektronix emulation mode. When using a real Tektronix 4010/4014 terminal, or when the "plotdevice" variable is set to "regis" or "dumb," no such sequences need be defined, although they may be for access to special features, such as to clear the screen before a plot. For example, to clear the screen of a VT100 terminal before plotting, and wait 10 seconds after plotting, the assignments would be:

```
preplot := [27,91,72,27,91,50,74];
postplot := [-10];
```

For further information, please consult the documentation accompanying your terminal.

9.1.83.13. plot/spline – explanation of SPLINE option

CALLING SEQUENCE : plot(f,h,v,style=SPLINE);

- There are three styles of plots available on the three types of plot devices. These are: POINT, LINE, SPLINE. The default is SPLINE.
- If the style is SPLINE, the points are interpolated by a natural cubic spline.

EXAMPLES :

```
plot([2,3,4,5,2,9],style=SPLINE);
```

plots a smooth line connecting the points

SEE ALSO : help(plot,`options`), help(plot,functions);

9.1.83.14. plot/structure – explanation of plot data structure

- A word about the way plotting in Maple is organized: The Maple function plot generates a list of points using Maple's arbitrary precision model of floating point arithmetic. A dummy "PLOT" data structure representing the plot is returned as the result of calling the plot function. The "PLOT" data structure is understood by Maple's pretty-printer which prints the plot in a "pretty" form just like it does any other Maple object. Thus, note the following
  1) lprint of a PLOT will print out the PLOT object

2) a PLOT can be saved in a file for later redisplay
3) a PLOT can be manipulated like any Maple object
The PLOT data structure is quite complicated. As mentioned above, its value can be seen by using lprint. Of particular interest is that it supports three different modes of plots and allows for the user to view a particular part of the plot. The modes are POINT (for plotting just the points), LINE (for linear interpolation between points) and SPLINE (for cubic spline interpolation between points – the default). To obtain a LINE plot simply substitute SPLINE = LINE in the PLOT data structure. To change the range on which the plot is displayed, include an argument of the form RANGE(VERTICAL,–2..2) in the plot data structure.

- The plot routine is "adaptive". It does a minimum of 25 function evaluations then does more work where f is changing most. Thus, it is not necessary to tell plot how many function evaluations to do.

EXAMPLES :

```
a := plot(sin,-Pi..Pi):                    assign a plot to a variable
a;                                         display plot
lprint(a);                                 display the PLOT data-structure
subs(SPLINE=POINT,a);                      change the style of the plot
b := plot([tan(t),t,0..1],coords=polar):   assign another plot
b;                                         display the second plot
save a,b,`myplots.m`;                      store your plots in a file
```

SEE ALSO : help(plot,`options`);

9.1.84. prem and sprem – pseudo-remainder and sparse pseudo-remainder of polynomials

CALLING SEQUENCES : prem(a,b,x,'m','q')
sprem(a,b,x,'m','q')

PARAMETERS :
- a, b – multivariate polynomials in the variable x
- x – indeterminate
- m, q – (optional) unevaluated names

- The function prem returns the pseudo-remainder "r" such that:

$$m\,a = b\,q + r$$

where degree(r,x) < degree(b,x) and "m" (the multiplier) is:

$$m = \mathrm{lcoeff}(b,x)\ \hat{}\ (\mathrm{degree}(a,x) - \mathrm{degree}(b,x) + 1)\ .$$

- If the fourth argument is present it is assigned the value of the multiplier "m" defined above. If the fifth argument is present, it is assigned the pseudo-quotient "q" defined above.

- The function sprem has the same functionality as prem except that it uses as a multiplier "m" the smallest possible power of lcoeff(b,x) such that the division process does not introduce fractions into q and r. When sprem can be used it is preferred because it is more efficient.

EXAMPLE :
```
a := x^4+1: b := c*x^2+1:
r := prem(a,b,x,'m','q'):
r,m,q;  yields  c*(c^2+1), c^3, c^2*x^2-c
r := sprem(a,b,x,'m','q'):
r,m,q;  yields  c^2+1, c^2, c*x^2-1
```
SEE ALSO : rem, quo, Prem, Sprem

9.1.85. Prem and Sprem – inert prem and sprem functions

CALLING SEQUENCES : Prem(a,b,x,'m','q')
Sprem(a,b,x,'m','q')

PARAMETERS :
- a, b – multivariate polynomials in the variable x
- x – indeterminate
- m, q – (optional) unevaluated names

- The "Prem" and "Sprem" functions are placeholders for the pseudo-remainder and sparse pseudo-remainder of "a" divided "b" where "a" and "b" are polynomials in the variable "x". They are used in conjunction with either mod or evala as described below which define the coefficient domain.
- The function Prem returns the pseudo-remainder "r" such that:
$$m\,a = b\,q + r$$
where degree(r,x) < degree(b,x) and "m" (the multiplier) is:
$$m = \mathrm{lcoeff}(b,x)^{(\mathrm{degree}(a,x) - \mathrm{degree}(b,x) + 1)}\ .$$
- If the fourth argument is present it is assigned the value of the multiplier "m" defined above. If the fifth argument is present, it is assigned the pseudo-quotient "q" defined above.
- The function Sprem has the same functionality as Prem except that it uses as a multiplier "m" the smallest possible power of lcoeff(b,x) such that the division process does not introduce fractions into q and r. When Sprem can be used it is preferred because it is more efficient.
- The calls Prem(a,b,x,'m','q') mod p and Sprem(a,b,x,'m','q') mod p compute the pseudo-remainder and sparse pseudo-remainder respectively of "a" divided "b" modulo "p", a prime integer. The coefficients of "a" and "b" must be multivariate polynomials over the rationals.

- The calls evala(Prem(a,b,x,'m','q')) and evala(Sprem(a,b,x,'m','q')) do likewise for "a" and "b" multivariate polynomials over the algebraic constants defined by RootOf's.

SEE ALSO : prem, sprem, mod, evala, RootOf

9.1.86. print – pretty-printing of expressions

CALLING SEQUENCE : print(e1, e2, ...);
PARAMETERS : e1, e2, ... – any expressions

- The function "print" displays the values of the expressions appearing as arguments, and returns NULL as the function value.
- The global variable 'prettyprint' is checked to determine the format in which the expressions are to be printed.
- If "prettyprint" is equal to 1 (the default value) the expressions will be displayed in a two-dimensional format, and they will be centered (if possible).
- If "prettyprint" equals two (i.e. prettyprint := 2;) the expressions will be displayed as for the case above except that they will always be left-justified.
- If "prettyprint" equals zero (i.e. prettyprint := 0;) the expressions will be line printed. (See 'lprint')
- The global variable 'screenwidth' specifies the number of characters that fit on the output device being used. Since most CRTs are 80 characters wide, the default value for 'screenwidth' is 79. This value can be changed by the user (e.g. screenwidth := 132; for wide printer output).
- "print" separates the expressions displayed by a comma and a blank.
- "print();" with no arguments will print a blank line.

SEE ALSO : lprint

9.1.87. product – product of a sequence

CALLING SEQUENCE : product(<expr>, i = m..n);
PARAMETERS :

	<expr>	– an algebraic expression
	i	– an unevaluated name
	m, n	– expressions

- if m and n are numeric constants, then "product" multiplies together all expressions formed by substituting "i" for the numbers: m, m+1 ,..., n in "expr".

- If m = n+1 then the value returned is 1. If m > n+1 then the value returned in 1/product(expr, i=n+1..m−1)

EXAMPLES :

```
product( 2*i^2, i=1..3 );     yields  288
product( i, i = 3/2..9/2 );   yields  945/16
product( i, i=n..n+2 )        yields  n (n+1) (n+2)
```

9.1.88. Psi – the polygamma function and its derivatives

The polygamma function is defined by

$$\text{Psi}(x) = \text{diff}(\ln(\text{GAMMA}(x)), x)$$

and its derivatives are represented by

$$\text{Psi}(n, x) = \text{diff}(\text{Psi}(x), x\$n) \ .$$

Automatic simplifications will be applied to these functions.

9.1.89. radsimp – simplification of an expression containing radicals

CALLING SEQUENCE : radsimp(<expr>, 'ratden');
PARAMETERS : <expr> – algebraic expression
ratden – optional unevaluated name

- "radsimp" simplifies radicals contained in <expr>.
- If the second argument, 'ratden', is present, it will be assigned the simplified expression with its denominator rationalized.

EXAMPLES :

```
radsimp( (1 + 2*x + x^2)^(1/2) );     yields  1 + x
radsimp( (1 + 2^(1/2))^(−1), 'd' );   yields  ( 1 + 2^(1/2) )^(−1)
d;                                    yields  −1 + 2^(1/2)
```

9.1.90. rand – random number generator

CALLING SEQUENCE : rand(r);
PARAMETERS : r – (optional) an integer range or an integer

- With no arguments, "rand();" returns a 24 digit non-negative random integer. With an integer range as an argument, "rand(a..b);" returns a procedure which, when called generates random integers in the range [a..b]. With an integer as an argument, "rand(n)" is short for rand(0..n−1).

- Because "rand(a..b)" returns a maple proc, the user can use more than one random number generator at the same time. However, since all random number generators use the same underlying random number sequence, calls to one random number generator will affect the random numbers returned from another.
- The global variable _seed can be used to alter the sequence of random numbers generated, by setting it to any non-zero integer.

EXAMPLES :

```
die := rand(1..6):   yields  a procedure
die();               yields  first throw of the die
die();               yields  second throw of the die
```

9.1.91. readlib – read a library file to define a specified name

CALLING SEQUENCE : readlib(f, file1, file2, ...):

PARAMETERS : f – the name of a procedure to be read
file1, ... – (optional) filenames

- With only one argument, "readlib(f):" causes the following statement to be executed:

  read ``.libname.`/`.f.`.m`;

  i.e., the ".m" file for f is loaded from the standard Maple library.
- In the case of two or more arguments, all arguments after the first are taken to be complete filenames of files to be read:

  read file1; read file2; . . . .
- In both cases, the value of the readlib function is the value assigned to the first argument "f" after executing the read statement(s).
- It is an error if "f" is not defined by this action.
- "readlib" has option remember; hence subsequent invocations of readlib with the same arguments will not cause reloading of the files.
- Many of Maple's library functions are initially "readlib-defined" in the system – e.g., the initial definition of the name "gcd" is:

  gcd := 'readlib('gcd')';

  Whenever the name "gcd" is evaluated, it causes the corresponding readlib to be executed, thus loading the gcd procedure.

EXAMPLE: readlib(randpoly):

SEE ALSO : libname, `read`, files, remember, help(index,libmisc)

9.1.92. readstat – read one statement from the input stream

CALLING SEQUENCE : readstat(<string>);
PARAMETER : <string> – a name

- The function "readstat" reads the next statement and returns the value of the statement.
- <string> is used as a prompt string, which is printed before maple's prompt.

EXAMPLE :

```
> a := readstat(`a will be assigned `);
a will be assigned >    456;
                        yields   a := 456
```

SEE ALSO : `read`

9.1.93. rem and quo – remainder / quotient of polynomials

CALLING SEQUENCE : rem(a,b,x); or rem(a,b,x,'q');
quo(a,b,x); or quo(a,b,x,'r');
PARAMETERS : a, b – polynomials in x
x – a name
q, r – (optional) unevaluated names

- The rem function returns the remainder of "a" divided by "b". The quo function returns the quotient of "a" divided by "b". The remainder "r" and quotient "q" satisfy: a = b*q + r where degree(r,x) < degree(b,x).
- If a fourth argument is present to rem or quo it will be assigned the quotient "q" or remainder "r" respectively.

EXAMPLE :

```
rem( x^3+x+1, x^2+x+1, x, 'q' );    yields   x + 2
q;                                  yields   x – 1
quo( x^3+x+1, x^2+x+1, x );         yields   x – 2
```

SEE ALSO : irem, iquo, prem, sprem, divide

9.1.94. Rem and Quo – inert rem and quo functions

CALLING SEQUENCES : Rem(a,b,x); or Rem(a,b,x,'q');
Quo(a,b,x); or Quo(a,b,x,'r');
PARAMETERS : a, b – polynomials in x
x – name (variable)
q, r – unevaluated name

- The "Rem" and "Quo" functions are placeholders for representing the remainder and quotient respectively of "a" divided "b" where "a" and "b" are polynomials in the variable "x" over a field. They are used in conjunction with either mod or evala as described below which define the coefficient domain.
- Functionality: Rem returns the remainder "r" and if the fourth argument "q" is present then the quotient is assigned to it. Quo returns the quotient "q" and if the fourth argument "r" is present then the remainder is assigned to it. The remainder "r" and quotient "q" satisfy: a = b*q + r .
- The calls Rem(a,b,x) mod p and Quo(a,b,x) mod p compute the remainder and quotient respectively of "a" divided "b" modulo "p", a prime integer. The coefficients of "a" and "b" must be multivariate polynomials over the rationals. In particular, if the coefficients are integers then the computation is done over the field of integers modulo "p".
- The calls evala(Rem(a,b,x)) and evala(Quo(a,b,x)) do likewise for "a" and "b" multivariate polynomials with algebraic extensions defined by RootOf's.

EXAMPLES :

```
a := x^4+5*x^3+6:
b := x^2+2*x+7:
Rem(a,b,x) mod 13;          yields  6+5*x
c := x^2-x+3:
d := x-RootOf(_Z^2-3):
evala(Quo(c,d,x));          yields  x-1+RootOf(_Z^2-3)
evala(Rem(c,d,x));          yields  6-RootOf(_Z^2-3)
```

SEE ALSO : rem, quo, mod, evala, RootOf

9.1.95. resultant – the resultant of two polynomials

CALLING SEQUENCE : resultant(a,b,x);

PARAMETERS : a, b – any two polynomials in x

- The function "resultant" computes the resultant of two polynomials with respect to the indeterminate x.
- Resultant takes advantage of any factorization that is present though no explicit factorization is attempted.

EXAMPLE :

```
resultant( a*x+b, c*x+d, x );      yields  a*d-b*c
resultant( (x+a)^5, (x+b)^5, x );  yields  (-a+b)^25
```

9.1.96. Resultant – inert resultant function

CALLING SEQUENCE : Resultant(a, b, x);
PARAMETERS : a,b polynomials
x main variable

- The "Resultant" function is a placeholder for representing the resultant of the polynomials "a" and "b" with respect to the main variable "x" . It is used in conjunction with either mod or evala as described below which define the coefficient domain.
- The call Resultant(a,b,x) mod p computes the resultant of "a" and "b" with respect to the main variable "x" modulo "p", a prime integer.
- The call evala(Resultant(a,b,x)) does likewise for "a" and "b" multivariate polynomials with algebraic extensions defined by RootOf's.

EXAMPLES :

```
Resultant(2*x+1, 3*x+4, x) mod 7;   yields  5
r := x + RootOf(_Z^2-2):
s := RootOf(_Z^2-2)*x + 1:
evala( Resultant(r, s, x) );        yields  -1
```

SEE ALSO : evala, mod, resultant

9.1.97. RETURN – explicit return from a procedure

CALLING SEQUENCE : RETURN(<expr1>, <expr2>, ...)
PARAMETERS : <expr1>, ... – expression sequence (possibly null)

- One common form of return from a procedure invocation occurs when execution "falls through" the end of the <statement sequence> which makes up the procedure body, in which case the value of the procedure invocation is the value of the last statement executed.
- A call to the RETURN function causes an immediate return to the point where the current procedure was invoked.
- If return is via the RETURN function then the value of the procedure invocation is the expression sequence specified in the call to RETURN.
- It is an error if a call to RETURN occurs at a point which is not within a procedure definition.

- A particular form of return is the "fail return", in which it is desired to return the unevaluated function invocation as the result. This can be done with the construct RETURN('procname(args)'). Here, "procname" and "args" are special names which are substituted by, respectively, the name with which the procedure was called, and the sequence of actual arguments with which the procedure was called.

SEE ALSO : procedures, procname, args, error

9.1.98. RootOf – a representation for roots of equations

CALLING SEQUENCE : RootOf(expr, x)
RootOf(expr1 = expr2, x)
RootOf(<expression in global variable _Z>)

PARAMETER :

expr	assumes an equation of the form expr=0
expr1=expr2	an algebraic equation
x	a variable name.

- The function RootOf is a place holder for representing roots of equations; in particular, it is the standard representation for Maple's algebraic numbers and algebraic functions.
- The RootOf function checks the validity of its arguments, and resolves it for polynomials of degree one. The RootOf is expressed in a single-argument canonical form obtained by making the argument primitive and expressing the RootOf in terms of the global variable _Z.
- Maple knows how to diff, taylor, evalf, and simplify RootOfs. Evaluation in the context of "evala" makes fundamental use of the RootOf notation for the representation of algebraic numbers and algebraic functions.

EXAMPLES :

RootOf(x^2+1=0,x);	yields	RootOf(_Z^2+1)
RootOf(x^2–a*x=b,x);	yields	RootOf(_Z^2–a*_Z–b)
RootOf(a*b^2+a/b,b);	yields	RootOf(_Z^3+1)
RootOf(a*x+b,x);	yields	–b/a

SEE ALSO : evala, convert(expr,RootOf)

9.1.99. rsolve – recurrence equation solver

CALLING SEQUENCE : rsolve(<equations(s)>, <function(s)>)

PARAMETERS :

<equations(s)> – a single equation or a set of equations; an expression is understood to be equated to 0; boundary conditions are given by providing equations for the specific

values of the function which are known (e.g. f(1) = 10, f(2) = a)

<function(s)> – f(n) or f(n,m);
indicates that the solution should be a general expression for f(n) (f(n,m) respectively); if only a name f is supplied then rsolve will attempt to solve for f(n) where n is determined by the indeterminates occurring as arguments to f

- Compute an expression for the nth term of the recurrence equation for f(n). Boundary conditions are provided by supplying extra equations of the form f(a) = A, f(b) = B.
- For difference equations of the form f(n) = a*f(n/b) + g(n), a method based on substitutions is used. If no boundary condition is given then f(1) = 0 is assumed.
- A second method based on characteristic equations is used for linear difference equations with constant coefficients. If no boundary conditions are given, the answer is expressed in terms of f(0), f(1), etc.
- The global names _N and _C are used during the solution process, but do not appear in the final result.

EXAMPLES :
```
rsolve( f(n) = -3*f(n-1) - 2*f(n-2) , f );
        (f(1) + 2*f(0))*(-1)^n + (- f(1) - f(0))*(-2)^n
rsolve( {f(n) = -3*f(n-1) - 2*f(n-2), f(1)=1, f(2)=1} , f );
                        - 3 (-1)^n + (-2)^n
rsolve( f(n) = 3*f(n/2) + 5*n, f(n) );
       n^(ln(3)/ln(2)) * (-15*(2/3)^(ln(n)/ln(2)+1) + 10)
```

9.1.100. sign – sign of a number or a polynomial

CALLING SEQUENCE : sign(<expr> , [x1, x2, ...] , 'y');
PARAMETERS : <expr> – a multivariate polynomial
[x1, ...] – (optional) list of indeterminates
y – (optional) unevaluated name

- "sign" computes the sign of <expr> in the sense: sign of the leading coefficient of <expr>.
- The leading coefficient of expr is determined with respect to the indeterminates given. If none are given the leading coefficient is taken with respect to all its indeterminates. Note therefore that the leading coefficient is dependent on the order of the indeterminates which may vary from one maple session to another, but not within a session.

- The unevaluated name is assigned the leading term.

EXAMPLES :

```
sign(expr);  is equivalent to sign(expr,indets(expr));
sign(expr,x); is    "     " sign(expr,[x]);
sign( x*y - x^2, [ x, y ] );                    yields  -1
sign( x*y - x^2, [ y, x ] , 'a' );              yields   1
a;                                              yields   x*y
sign( 0 );                                      yields   1
sign( -2/3 );                                   yields  -1
```

SEE ALSO : indets, lcoeff, signum

9.1.101. signum – sign function for real and complex expressions

CALLING SEQUENCE : signum(z);
PARAMETER : z – any algebraic expression

- "signum" computes the sign of a real or complex number.
- For rationals and floats, it returns –1 if the argument is negative; otherwise it returns 1 . Note that signum(0) is defined to be 1 . For a complex number argument, signum(z) returns z/abs(z) .
- "signum" maps onto products, uses 10-digit floating-point arithmetic for constants, and performs a few other simplifications.

SEE ALSO : sign

9.1.102. simplify – apply simplification rules to an expression

CALLING SEQUENCE : simplify(e); or simplify(e,n1,n2,...);
PARAMETERS : e – any expression
n1,n2,... – (optional) names

- The simplify function is used to apply simplification rules to an expression. If only one argument is present, then simplify will search the expression for function calls, square roots, radicals, and powers. Next it will invoke the appropriate simplification procedures, which include: GAMMA, RootOf, ln, exp, sqrt, trig (for trig functions), hypergeom (for hypergeometrics), radical (occurrence of fractional powers), power (occurrence of powers, exp, ln), and atsign (for operators).
- For further information on a particular simplification procedure, do: help(simplify,<procedure>); where <procedure> is one of the simplification procedures listed above.

- In the case of two or more arguments, then simplify will only invoke the simplification procedures specified by the additional arguments.
- A user can make his own simplifications known to the simplify function by defining a maple procedure. If the procedure `simplify/f` is defined, then the function call simplify(a,f); will invoke `simplify/f`(a);

EXAMPLES :

```
simplify( cos(Pi/4)^5 + sin(Pi/4)^4 );    yields 1/8*2^(1/2)+1/4
simplify( 4^(1/2)+3 );                    yields 5
simplify( (x^a)^b+4^(1/2), power );       yields x^(a*b)+4^(1/2)
simplify( exp(a+ln(b*exp(c))) );          yields b*exp(a+c)
simplify( sin(x)^2+cos(x)^2, trig );      yields 1
```

SEE ALSO: normal, radsimp, expand, factor

9.1.102.1. simplify/atsign – simplify expressions involving operators

CALLING SEQUENCE : simplify(<expr>, atsign);
PARAMETERS : <expr> – any expression

- The simplify/atsign function is used to simplify expressions that contain operators.
- It simplifies compositions of inverses.

EXAMPLES :

```
simplify( sin@arcsin, atsign );    yields <x | x>
simplify( exp@log, atsign );       yields <x | x>
```

SEE ALSO: `@`, help(operators,functional)

9.1.102.2. simplify/GAMMA – simplifications involving the GAMMA function

CALLING SEQUENCE : simplify(<expr>, GAMMA);
PARAMETERS : <expr> – any expression

- The simplify/GAMMA function is used to simplify expressions containing then gamma function.

EXAMPLES :

```
simplify( GAMMA(-1), GAMMA );   yields  Error
simplify( GAMMA(6), GAMMA );    yields  120
simplify( GAMMA(3/2), GAMMA );  yields  1/2*Pi^(1/2)
```

SEE ALSO: GAMMA, simplify

9.1.102.3. simplify/hypergeom – simplify expressions involving hypergeometrics

CALLING SEQUENCE : simplify(<expr>, hypergeom);
PARAMETERS : <expr> – any expression

- The simplify/hypergeom function is used to simplify expressions that contain hypergeometric functions.

EXAMPLES :

```
simplify( hypergeom([-1],[1],z), hypergeom );   yields  1-z
```

SEE ALSO: simplify, hypergeom

9.1.102.4. simplify/power – simplify powers

CALLING SEQUENCE : simplify(<expr>, power);
PARAMETERS : <expr> – any expression

- The "simplify/power" procedure is used to simplify powers, exponentials, and logarithms in an expression.

EXAMPLES :

```
simplify( (a^b)^c, power );      yields  a^(b*c)
simplify( x^a*x^b, power );      yields  x^(a+b)
simplify( exp(5*ln(x)), power ); yields  x^5
simplify( ln(x*y), power );      yields  ln(x)+ln(y)
```

SEE ALSO: simplify

9.1.102.5. simplify/Psi – simplify expressions with the Psi function

CALLING SEQUENCE : simplify(<expr>, Psi);
PARAMETERS : <expr> – any expression

- The simplify/Psi function is used to simplify expressions that contain the Psi function.

- When simplify/Psi is executed, the Psi routine is read in so any future expressions containing Psi will automatically be simplified.

EXAMPLES :

```
simplify( Psi(1/2), Psi );   yields  -gamma-2*ln(2)
Psi(5);                      yields  25/12-gamma
```

SEE ALSO: simplify, Psi

9.1.102.6. simplify/radical – apply radical simplifications to an expression

CALLING SEQUENCE : simplify(<expr>, radical);
PARAMETERS : <expr> – any expression

- The simplify/radical function is used to simplify expressions that contain radicals.
- This routine is just a call to radsimp.

EXAMPLES :

```
simplify( (x-2)^(3/2)/(x^2-4*x+4)^(1/4), radical );   yields  x-2
```

SEE ALSO: radsimp, simplify

9.1.102.7. simplify/RootOf – simplify expressions with the RootOf function

CALLING SEQUENCE : simplify(<expr>, RootOf);
PARAMETERS : <expr> – any expression

- The simplify/RootOf function is used to simplify expressions that contain the RootOf function.
- Polynomials and inverses of polynomials in RootOf are simplified. Any n-th roots involving RootOf are resolved.

EXAMPLES :

```
r := RootOf(x^2-2=0,x):
simplify( r^2, RootOf );      yields  2
simplify( 1/r, RootOf );      yields  1/2*RootOf(x^2-2=0,x)
```

SEE ALSO: RootOf, simplify

9.1.102.8. simplify/sqrt – simplify square roots

CALLING SEQUENCE : simplify(<expr>, sqrt);
PARAMETERS : <expr> – any expression

- The simplify/sqrt function is used to simplify expressions that contain square roots or powers of square roots.
- It extracts any square integer or polynomial factor from underneath the square root.
- The positive square root is used.

EXAMPLES :

```
simplify( 16^(3/2), sqrt );                    yields 64
simplify( (10*x^2+60*x+90)^(1/2), sqrt );      yields 10^(1/2)*(x+3)
simplify( (90/121)^(1/2), sqrt );              yields 3/11*10^(1/2)
```

SEE ALSO: simplify

9.1.102.9. simplify/trig – simplify trigonometric expressions

CALLING SEQUENCE : simplify(t, trig);
PARAMETERS : t – any expression

- The simplify/trig function is used to simplify trigonometric expressions by applying the identities sin(x)^2+cos(x)^2=1 and cosh(x)^2-sinh(x)^2=1.
- Simplify/trig views the input as a polynomial in sin(x) and sinh(x) and reduces the degree of the input to 1 in sin(x) and sinh(x).

EXAMPLES :

```
simplify( sin(x)^3, trig );          yields sin(x) – sin(x)*cos(x)^2
simplify( tan(x)^2, trig );          yields – (cos(x)^2 – 1) / cos(x)^2
simplify( 1+tan(x)^2, trig );        yields 1 / cos(x)^2
simplify( cos(2*x)+sin(x)^2 );       yields cos(x)^2
```

SEE ALSO: simplify

9.1.103. solve – solve equations

CALLING SEQUENCE : solve(<eqns>, <vars>);
PARAMETERS : <eqns> – an equation or set of equations
<vars> – an unknown or set of unknowns

- The most common application of solve is to solve a single equation, or to solve a system of equations in some unknowns. A solution to a single equation <eqns> solved for the unknown <vars> is returned as an expression. To solve a system of equations <eqns> for unknowns <vars>, the system is specified as a set of equations and a set of unknowns. The solution is returned as a set of equations.

- Multiple solutions are returned as an expression sequence. Wherever an equation is expected, if an expression <expr> is specified then the equation <expr> = 0 is understood. If <vars> is not specified, indets(<eqns>,name) is used.
- When solve is unable to find any solutions, the expression NULL is returned. This may mean that there are no solutions or that solve was unable to find the solutions.
- To assign the solutions to the variables, use the command "assign".
- For solving differential equations use "dsolve", for purely floating point solutions use "fsolve", use "isolve" to solve for integer roots, "msolve" to solve modulo a prime, and "rsolve" for recurrences.
- For further help do: help(solve,<subtopic>); where <subtopic> is one of: floats, functions, identity, ineqs, linear, radical, scalar, series or system.

EXAMPLES :

solve(cos(x) + y = 9, x);	yields	arccos(−y + 9)
solve({cos(x) + y = 9}, {x});	yields	{x = arccos(−y + 9)}
solve(x^3 − 6*x^2 + 11*x − 6, x);	yields	1, 2, 3
solve({x+y=1, 2*x+y=3}, {x,y});	yields	{x=2,y=−1}
solve({a*x^2*y^2, x−y−1}, {x,y});	yields	{x=1,y=0}, {x=0,y=−1}

SEE ALSO : dsolve, fsolve, isolve, msolve, rsolve, assign, isolate, match, simplex and grobner.

9.1.103.1. solve/float – solve in the presence of floating-point numbers

CALLING SEQUENCE : solve(<eqns>, <vars>)

PARAMETERS : <eqns> equations (as for solve), but with floating values
<vars> variables (as for solve)

- The solve function with floating-point numbers works by converting the floating-point numbers to approximate rationals, calling solve with these converted arguments, and converting the results back to floating-point numbers.
- This is useful for solving equations with a combination of floating-point numbers and parameters (since fsolve will not solve equations with unassigned parameters).

EXAMPLES :

```
solve( x^2-3*x+0.01 );                yields 2.996662955, .003337045
solve( {x^2-y=z, 3.7*y+z=sin(x)} );   yields
            {x=x, y=-.3703704*x^2+.3703704*sin(x),
                z=1.370370*x^2-.3703704*sin(x)}
```

SEE ALSO : fsolve, evalf, realroots

9.1.103.2. solve/function – solve for a function

CALLING SEQUENCE : solve(<eqn>, <fcnname>)
PARAMETERS : <eqn> a single equation (as for solve)
<fcnname> a single variable (as for solve)

- When the unknown is a name which is only used as a function in the equation, the solver constructs a function or procedure which solves the equation.

EXAMPLES :

```
solve( cos(f(x))=sin(x), f );      yields proc (x) arccos(sin(x)) end
solve( f(x)^2-3*f(x)+2*x, f );     yields
proc (x) 3/2+1/2*(9-8*x)^(1/2) end, proc (x) 3/2-1/2*(9-8*x)^(1/2) end
```

SEE ALSO : unapply, help(operators,functional)

9.1.103.3. solve/identity – solve expressions involving identities

CALLING SEQUENCE : solve(identity(<eqn>, <x>), <vars>)
PARAMETERS : <eqn> an equation, or an expression (equated to 0)
<x> the identity variable
<vars> variables to be determined (as for solve)

- The expression (or equation) <eqn> is considered an identity in terms of the variable <x>, and solve attempts to find a solution in terms of <vars> which satisfies the equation <eqn> for any value of <x>.
- This is a one-dimensional case of a generalized pattern matching.
- <x> should not be one of the <vars>.

EXAMPLES :

```
solve( identity( sin(x)=cos(a*x+b), x ), {a,b} );                yields
                                {b=Pi/2, a=-1}
solve( identity( 3/(x^2+4*x+4)=A*(x+B)^P, x ), {A,B,P} );        yields
                              {P=-2, A=3, B=2}
solve( identity( y/x = a*(x+c)^b, x ), {a,b,c} );                yields
                              {b=-1, c=0, a=y}
```

SEE ALSO : match

9.1.103.4. solve/ineq – solve inequalities

CALLING SEQUENCE : solve(<ineq>, <var>)
PARAMETERS : <ineq> one inequality ("<", "<=", ">" or ">=")
<var> one variable (as in solve)

- Solve will solve one inequality in one variable for all the cases where the equality points of <ineq> can be properly ordered.
- Some additional cases can be handled with the use of the signum function.

EXAMPLES :

```
solve( a*x < b, x );                          {signum(a)*x < signum(a)*b/a}
solve( a^2*x^2 >= 0, x );                     {x = x}
solve( x+1/x > 0, x );                        {0 < x}
solve( (x-1)*(x-2)*(x-3) < 0, x );            {x < 1}, {2 < x, x < 3}
solve( (x-1+a)*(x-2+a)*(x-3+a) < 0, x );
                                              {x < 1-a}, {2-a < x, x < 3-a}
solve( exp(x) > x+1 );                        {x <> 0}
```

SEE ALSO : simplex, isolate

9.1.103.5. solve/linear – solve systems of linear equations

CALLING SEQUENCE : solve(<eqns>, <vars>)
PARAMETERS : <eqns> a set of linear equations and inequations
<vars> a set of (at least two) names (unknowns)

- Inequations (expr1 <> expr2) can be part of the input. It is guaranteed that the solutions will satisfy the equations and inequations.
- If there are more unknowns than equations, the solver will select the variables for which the system appears easiest to solve.
- Currently, the solver uses four separate algorithms. All algorithms are geared to large sparse systems though they also perform well on dense systems.
- The special algorithms are selected automatically and are for: Rationals, using a "primitive" fraction-free algorithm; Floats, using Gaussian elimination with pivoting for stability; Q(x1,x2,...,xn), using a "primitive" fraction-free algorithm and Complex numeric, using Gaussian elimination.

EXAMPLES :

```
solve( {a*x+b*y=3,x-y=b}, {x,y} );  yields
      {x = (b^2 + 3) / (a + b), y = - (a*b - 3) / (a + b)}
```

SEE ALSO : help(solve,system), help(linalg,linsolve)

9.1.103.6. solve/radical – solve expressions involving radicals

CALLING SEQUENCE : solve(<eqns>, <vars>)
PARAMETERS : <eqns> equations (as for solve), but with radicals
<vars> variables (as for solve)

- The solve function will solve equations with radicals using new auxiliary variables and adding new equations. The resulting problem is always a system of equations, and hence it is solved like a system and not as a scalar.
- Radicals will be considered to be multivalued functions, and hence verification of the solution requires finding the right combination of branches of the radicals.

EXAMPLES :

```
                                                    yields:
solve( sqrt(x+1)-sqrt(x-1)=a, x );                  1/4*(4+a^4)/a^2
solve( x+sqrt(x)-2, x );                            1, 4
solve( x+sqrt(x)+x^(1/3)=3, x);
        1, RootOf(_Z^5+_Z^4+_Z^3+2*_Z^2+3*_Z+3)^6
solve( sqrt(x)+sqrt(x+1)=3 );                       16/9
solve( {sqrt(x)+sqrt(y)-2, x-y+1}, {x,y});          {y = 25/16, x = 9/16}
```

SEE ALSO : help(solve,systems), help(solve,scalar)

9.1.103.7. solve/scalar – solve a single equation for a single variable

CALLING SEQUENCE : solve(<eqn>, <var>)
PARAMETERS : <eqn> algebraic expression (to be equated to 0), or equation or set containing a single expression
<var> optional name, or set containing a single name

- The solve function with a single equation and a single variable uses the scalar solver. When called with a single argument, it is equivalent to solve(<eqn>, indets(<eqn>,name)).
- The scalar solver will report all solutions found, even if some roots are multiple roots. In particular, for polynomials, solve is guaranteed to find all solutions.
- For polynomials of degree 2, 3 and 4, the scalar solver will compute the explicit solutions. For irreducible and non-composable polynomials of degree higher than 4, the scalar solve will return the solutions using RootOf.

- Solve normally returns an expression sequence of sets, but when called without sets, i.e. an equation or an expression and a variable, it returns an expression sequence of the solutions.

EXAMPLES :

```
solve( cos(x) + y = 9, x );          yields  arccos(-y + 9)
solve( x^3 - 6*x^2 + 11*x - 6, x );  yields  1, 2, 3
solve( {x^3 - 6*x^2 + 11*x - 6}, {x} );  yields  {x = 1}, {x = 2}, {x = 3}
```

SEE ALSO : help(solve,floats), help(solve,identity), help(solve,radical)

9.1.103.8. solve/series – solve expressions involving general series

CALLING SEQUENCE : solve(<eqn>, <var>)

PARAMETERS : <eqn> an equation involving a series in <var>
<var> the variable to be solved for

- Solving an equation which contains a series for the main series-variable achieves a generalized inversion of series.
- The result, typically a series, will be in one of the remaining indeterminates. The indeterminate which produces the simplest answer is usually chosen.
- The global variable Order will be used to determine the order of the result, usually a series.
- Formally, if t = solve(f(series_in_x,y), x) is a series in y with sufficient terms, then substituting t for x in f will result in 0 or O(y^Order).

EXAMPLES :

```
Order := 3;
solve( taylor(x*exp(x),x) = y, x )   yields y - y^2 + O(y^3)
solve( taylor(exp(x),x) = a+b )      yields
    {x = b+a-1 - 1/2*(b+a-1)^2 + O((b+a-1)^3)}
```

SEE ALSO : taylor, RootOf

9.1.103.9. solve/system – solve systems of equations

CALLING SEQUENCE : solve(<eqns>, <vars>)

PARAMETERS : <eqns> a set of equations and inequations
<vars> a set of (at least two) names (unknowns)

- The system solver behaves slightly differently than the scalar solver. For linear systems of equations see help(solve,linear).
- Multiple solutions are reported only once. Solutions of irreducible polynomials of degree 2, 3 or 4 are represented implicitly by RootOfs.

- Inequalities (expr1 <> expr2) are either part of the input or are generated from denominators, singularities, etc. It is guaranteed that the solutions will satisfy the equations and inequations.
- If there are more unknowns than equations, the solver will select the variables for which the system is easiest to solve.
- If the global variable _MaxSols is set to an integer value, the solver will stop after _MaxSols solutions are found. By default this value is set to 100.

EXAMPLES :

```
solve( {x+2*y=3, y+1/x=1}, {x,y} );    yields
                {x=-1, y=2}, {x=2, y=1/2}
solve( a*x^2+b*x+c, {a,b,c,x} );        yields
            {c=-a*x^2-b*x, x=x, a=a, b=b}
solve( {x^2-y^2-y, x+y^2, x<>0} );    yields
{y = RootOf(_Z^3-_Z-1), x = - RootOf(_Z^3-_Z-1)^2}
```

SEE ALSO : help(solve,linear)

9.1.104. sort – sort a list with a specified ordering

CALLING SEQUENCE : sort(L , predfunc); or sort(L);

PARAMETERS : L – a list to be ordered
predfunc – a boolean procedure of two arguments

- "sort" sorts the elements of the list L using "predfunc" to determine the ordering. predfunc(a,b) should return true if a precedes b in the desired ordering, false otherwise.
- If the second argument is `<` or `<=` or numeric then it sorts numbers in increasing order.
- If the second argument is lexorder or string then it sorts strings in lexicographic order.
- If the second argument is address then "address" order is used.
- If only one argument is given then sort will choose one of the orderings `<` , lexorder or address.
- The sorting algorithm used is Shellsort.

EXAMPLES :

```
sort([c, a, d, aa, c], lexorder );  yields  [a, aa, c, c, d]
sort([3, 2, 1]);  yields  [1, 2, 3]
```

9.1.105. sqrt – square root

CALLING SEQUENCE : sqrt(x)
PARAMETERS : x – any algebraic expression

- The function sqrt forms the square root of its argument x. It returns the square root of integers and polynomials which are perfect squares. It is mapped onto products.

EXAMPLES :

```
sqrt(3);        yields  3^(1/2)
sqrt(4);        yields  2
sqrt(12);       yields  2*3^(1/2)
sqrt(3+4*I);    yields  2+I
sqrt(x^2/9);    yields  x/3
sqrt(a);        yields  a^(1/2)
```

9.1.106. subs – substitute subexpressions into an expression

CALLING SEQUENCE : subs(s1, s2, ..., sN, <expr>);
PARAMETERS : s1, ... – equations or sets or lists of equations
<expr> – any expression

- "subs" returns an expression resulting from applying the substitutions specified by the first arguments to the last argument <expr>.
- The substitutions are performed sequentially starting with "s1". The substitutions within a set or list are performed simultaneously.
- Every occurrence of the left hand side of a substitution equation that appears in <expr> is replaced by the right hand side of the equation.
- Note that only sub-expressions in <expr> that correspond to op's of a maple object are matched. This is termed "syntactic substitution".

EXAMPLES :

```
subs( x=r^(1/3), 3*x*ln(x^3) );   yields  3*r^(1/3)*ln(r)
subs( a+b=y, (a+b+c)^2 );         yields  (a+b+c)^2
subs( x=y, y=x, [x,y]);           yields  [x,x]
subs( {x=y, y=x}, [x,y]);         yields  [y,x]
subs( sin(x)+cos(x) );            yields  sin(x)+cos(x)
```

SEE ALSO : op, subsop

9.1.107. subsop – substitute for specified operands in an expression

CALLING SEQUENCE : subsop(eq1, eq2, ..., eqN, <expr>);
PARAMETERS : eqI – an (optional) equation of the form:
<numI> = <exprI>
<numI> – a non-negative integer
<exprI> – any expression
<expr> – an expression

- "subsop" will do simultaneous substitution with respect to the equation arguments in the last argument <expr>.
- "subsop" will return the expression resulting from replacing op(i1, <expr>) by <a1>, and op(i2, <expr>) by <a2>, and ..., op(iN, <expr>) by <aN> in <expr>.

EXAMPLES :

```
subsop( 3=y^4, x^7+8*x^6+x-9 );     yields  x^7+8*x^6+y^4-9
subsop( 1=NULL, 2=b, [x,y,x] );     yields  [b,x]
subsop( x^2+5 );                    yields  x^2+5
subsop( 0=g, 3=a, f(x,f(x,y,z),x) ); yields  g(x,f(x,y,z),a)
```

SEE ALSO : subs, op

9.1.108. substring – extract a substring from a string

CALLING SEQUENCE : substring(<string>, m..n);
PARAMETERS : <string> – a string
m..n – a range

- If "m" and "n" evaluate to integers then "substring" will extract a substring from <string> starting with the m'th character and ending with the n'th character. If m is less than one then the substring will start at the first character of <string>. If n is greater than the length of <string> then the substring will return a string from the m'th character to the end of <string>. If m is greater than n then substring will return the null string.
- If either "m" or "n" fails to evaluate to an integer, then "substring" will remain unevaluated.

EXAMPLES :

```
substring( abcdefgh, 3..7 );   yields  cdefg
substring( wxyz, 2..2 );       yields  x
```

SEE ALSO : length, cat

9.1.109. sum – definite and indefinite summation

CALLING SEQUENCE : sum(f, i); sum(f, i=m..n);
PARAMETERS : f – an expression
i – an unevaluated name
m, n – integers or arbitrary expressions

- The function "sum" will compute the sum of a sequence f(i) over a specified range or else as an indefinite summation.
- If the second argument to "sum" is "i" then the indefinite summation is computed. I.e., an expression (say "g") is returned such that:
  g(i+1) – g(i) = f(i) for all "i".
- If the second argument to "sum" is "i=m..n" then the definite summation is computed with respect to "i". I.e., the sum is:
  f(m) + f(m+1) + ... + f(n).
  If m = n+1 then the sum is 0.
- The definite sum is equivalent to g(n+1) – g(m) where "g" is the indefinite sum. The sum is computed in this way if "m" and "n" are not integers or if m–n is a large integer.
- Note: It is recommended (and often necessary) that both 'f' and 'i' be enclosed in single quotes to prevent premature evaluation. (E.g., i may have had a previous value). I.e., the common format is sum('f', 'i'=m..n);

EXAMPLES :

sum(i^2, i=1..3);	yields	14
sum(A[i], i = 1..3);	yields	A[1] + A[2] + A[3]
sum(i^2+i, i);	yields	1/3*i^3 – 1/3*i
sum(1/(i+1), i=1..n);	yields	Psi(n+2) – Psi(2)

9.1.110. system – invoke a command in the host operating system

CALLING SEQUENCE : system(<command>);
PARAMETER : <command> – a maple string

- "system" passes <command> to the host operating system (for example UNIX, VMS, CMS, etc.) which performs the appropriate function, and then the return status of the command is returned as the value of system.
- Note: On some operating systems, some operating system commands may not be accessible via this mechanism.

EXAMPLE : (For Maple being run under UNIX)
system(date); prints the date, and yields 0
SEE ALSO : escape, help(`option`) for "option system"

9.1.111. table – create a table

CALLING SEQUENCE : table(<indexfcn>, <list>);
PARAMETERS : <indexfcn> – (optional) an indexing function
<list> – (optional) list of initial table values

- A table is created either explicitly by a call to the "table" function or implicitly by assigning to an indexed name. Indices are arbitrary.
- Only <indexfcn> may be extracted by the "op" function. The <list> of values is accessible only via indexing (e.g. T[1,2]).
- The "table" function will set up a table with initial values specified by <list>, which is a list of equations each with left-hand–side a parenthesized sequence of indices and right-hand–side the value.
- If the <list> is specified simply as a list of values then the indices are taken to be 1, 2, ... etc. If no <list> is specified then the table is created with no values assigned.
- <indexfcn> can be a procedure or a name specifying how indexing is to be performed; if null then ordinary indexing is implied. Known names are: symmetric, antisymmetric, sparse, sparsymm, diagonal, identity.
- Tables have special evaluation rules (like procedures) so that if the name T has been assigned a table then T evaluates to T; op(T) yields the actual table structure; op(op(T)) yields <indexfcn>.
- Elements may be added to a table via the indexing notation. Thus T := table([4]); is equivalent to: T := table(); T[1] := 4; or equivalently: T[1] := 4; if T was unassigned (implicit creation).

EXAMPLES :

table();	yields table([])
table([22,42]);	yields table([(1)=22,(2)=42])
S := table([(2)=45,(4)=61]);	yields S := table([(2)=45,(4)=61])
S[1], S[2];	yields S[1], 45
table(symmetric,[(c,b,c)=x]);	yields table([(b,c,c)=x])
T[x,cos(x)] := tan(x): op(T);	yields table([(x,cos(x))=tan(x)])

SEE ALSO : indexed, array, symmetric, antisymmetric, sparse, sparsymm, diagonal, identity, indexfcn, indices, copy, selection

9.1.112. taylor – Taylor (Laurent) series expansion

CALLING SEQUENCE : taylor(<expr>, <eq/nm>, n);
PARAMETERS : <expr> – an expression
<eq/nm> – an equation (i.e. x = a) or name (i.e. x)
n – (optional) non-negative integer

- "taylor" computes a truncated series expansion of <expr>, with respect to the variable "x", about the point "a", up to order "n".
- If <eq/nm> evaluates to a name "x" then the equation "x=0" is assumed.
- If the third argument "n" is present then it specifies the "truncation order".
- If "n" is not present, the "truncation order" is determined by the global variable Order. The user may assign any non-negative integer to Order. The default value of Order is 6.
- If the series is not exact, then an "order term" (for example O(x^6)) is the last term in the series.
- For a complete list of known functions, see help(inifcns);
- "taylor" has an interface for user-specified functions. For example, if the procedure `taylor/f` is defined, then the function call taylor(f(x,y), x); will invoke `taylor/f`(x,y,x); to compute the series.
- If "taylor" is applied to an unevaluated int then the series expansion of the integral will be computed (if possible).

EXAMPLES :

```
taylor( 1/x + y + x^3, x=0 );     yields x^(-1) + y + x^3
taylor( 1/x, x=1, 3 );            yields
          1 - (x-1) + (x-1)^2 + O((x-1)^3)
taylor( exp(x), x=0, 4);          yields
          1 + x + 1/2*x^2 + 1/6*x^3 + O(x^4)
taylor( GAMMA(x), x=0, 3);        yields
x^(-1) - gamma + (1/12*Pi^2 + 1/2*gamma^2)*x + O(x^2)
int( cos(exp(x)), x );            yields  Ci(exp(x))
taylor(", x=0, 2);                yields
          (gamma + Ci(1)) + cos(1)*x + O(x^2)
```

SEE ALSO : inifcns, series

9.1.113. testeq – random polynomial-time equivalence tester

CALLING SEQUENCE : testeq(a = b); testeq(a, b); testeq(a);
PARAMETERS : a, b – arbitrary expressions

- The function testeq tests for equivalence probabilistically. It returns "false" if the expressions are not equal (or not equal to 0) and true otherwise for the class of expressions that testeq understands. The result "false" is always correct; the result "true" may be incorrect with very low probability.
- Testeq will succeed over expressions formed with rational constants, independent variables and I, combined by arithmetic operations, exponentials, trigonometrics and a few others. Testeq may also succeed with some expressions involving radicals, Pi as an argument of trigonometrics, and algebraic constants and functions. If the expressions do not fall in this class, testeq returns FAIL. Testeq may also return FAIL if it cannot find an appropriate modulus to work after 7 trials.
- Based on: "Determining Equivalence of Expressions in Random Polynomial Time" by Gaston Gonnet, Proceedings of the 16th ACM Symposium on the Theory of Computing, Washington DC, April 1984, pp. 334–341.

EXAMPLES :

```
a := (sin(x)^2 - cos(x)*tan(x)) * (sin(x)^2 + cos(x)*tan(x))^2:
b := 1/4*sin(2*x)^2 - 1/2*sin(2*x)*cos(x) - 2*cos(x)^2
     + 1/2*sin(2*x)*cos(x)^3 + 3*cos(x)^4 - cos(x)^6:
evalb( a = b );                    yields  false
evalb( expand(a) = expand(b) );    yields  false
testeq( a = b );                   yields  true
```

9.1.114. time – total CPU time used for the session

CALLING SEQUENCE : time();
PARAMETERS : none

- The function "time" returns the total CPU time used since the start of the Maple session. The units are in seconds and the value returned is a floating-point number.
- To time particular statements or groups of statements, one method would be the following:

```
st := time():
. . . statements to be timed . . .
time() - st;
```

Note the use of a colon to prevent the display of an uninteresting value; it is the value time() – st which gives the desired information.

SEE ALSO : profile, showtime, 'status', words

9.1.115. trace – tracing procedures as a debugging tool

CALLING SEQUENCES : trace(f); or trace(f,g,h,...);
untrace(f); or untrace(f,g,h,...);

PARAMETERS : f,g,h,... – name(s) of procedure(s) to be traced

- The trace function is a debugging tool for tracing the execution of the procedure(s) f, g, h, etc.
- During execution, the entry points, the statements that are executed, and the exit points of the traced procedure(s) are printed. At entry points, the actual parameters are displayed. At exit points, the returned function value is displayed.
- The untrace function turns off the tracing of its arguments.
- The trace (untrace) function returns an expression sequence of the names of the procedures for which tracing was turned on (off).

SEE ALSO : profile, help(linalg,trace)

9.1.116. traperror – trap an error condition

CALLING SEQUENCE : traperror(<expr1>, <expr2>, ...);
PARAMETERS : <expr1>, <expr2>, ... – any expressions

- "traperror" does the following:

(1) It sets an error trap prior to evaluating its arguments.

(2) If an error occurs during either evaluation or simplification, traperror returns the string corresponding to the error message of the first error occurring.

(3) Otherwise, it simply returns its simplified evaluated arguments.

- The most recently occurring error is stored in the variable lasterror. This variable can be manipulated as any other variable. "lasterror" is useful for checking the results of traperror as in the following:

```
if traperror(...) = lasterror then ...
```

Note: "traperror" and "lasterror" must appear in this order.

9.1.117. trigonometric functions

The following function names are used: **sin, cos, tan, csc, sec, cot.** Automatic simplifications will be applied to these functions.

9.1.118. trunc, round, and frac – truncate / round a number to an integer

CALLING SEQUENCE : trunc(x); and frac(x); and round(x);
PARAMETER : x – any expression

- "trunc" truncates a numeric argument down to an integer. "round" rounds a numeric argument up to an integer. "frac" computes the fractional part of a number. It is defined as "trunc(x)–x". All functions return unevaluated if their arguments are not numbers.

EXAMPLES :

```
trunc( 7 );      yields  7
trunc( 8/3 );    yields  2
round( 8/3 );    yields  3
frac( 8/3 );     yields  2/3
trunc( –2.4 );   yields  –2
trunc( x );      yields  trunc(x)
```

9.1.119. type – type-checking function

CALLING SEQUENCE : type(x,t); or type(x,t,...)
PARAMETERS : x – any expression
t – type name, or set or list of type names

- The type function is a boolean function which returns true if "x" is of type "t"; false otherwise.
- In the case where "t" is specified as a set or list of type names, the value returned is true if "x" is any of the types in the set/list; false otherwise.
- If more than two arguments are specified then the additional arguments are passed on to the specific type-checking function `type/t`.
- The following type names are known to Maple:

`=`	`<>`	`<`	`<=`	`.`	`..`
`!`	`+`	`*`	`^`	`**`	`and`
`or`	`not`	`union`	`intersect`	`minus`	algebraic
algfunc	algnum	array	boolean	constant	cubic
equation	even	expanded	float	fraction	function
indexed	integer	laurent	linear	list	listlist
logical	mathfunc	matrix	monomial	name	negative
nonneg	numeric	odd	point	polynom	positive
procedure	quadratic	quartic	radical	range	rational
ratpoly	relation	RootOf	scalar	series	set
sqrt	string	table	taylor	trig	uneval
vector					

- A user can make a datatype known to the type function in the following way. If the user has defined the procedure `type/mytype` then a call to type of the form: type(a,mytype,b,c,...); will generate the function call `type/mytype`(a,b,c,...).
- For further information, do help(type,<datatype>); where <datatype> is one of: algfunc, algnum, boolean, constant, cubic, expanded, laurent, linear, listlist, matrix, monomial, point, polynom, quadratic, quartic, radical, ratpoly, scalar, taylor, trig, or vector.
- Other <datatype>s in the above list are "fundamental types" for which help is available directly via help(<datatype>);

SEE ALSO : hastype, whattype, help(index,datatypes), convert

9.1.119.1. type/algfunc – check for algebraic function

CALLING SEQUENCE : type(e,algfunc); or type(e,algfunc,v);
PARAMETERS : e – an expression
v – variable(s)

- An expression "e" is of type algfunc (algebraic function) if it is a rational function in the variable(s) "v" extended by radicals.
- The variable(s) "v" is an indeterminate or a list or set of indeterminates. If no variables were specified, the indeterminates of "e" which are names are used. That is, "e" must be an algebraic function in all of its variables.

EXAMPLES:

```
type( x/(1-x), algfunc, x );          yields true
type( sqrt(x-y), algfunc );           yields true
type( sqrt(x-y), algfunc, x );        yields true
type( sqrt(x-sin(y)), algfunc );      yields false
type( sqrt(x-sin(y)), algfunc, x );   yields true
type( sqrt(x-sin(y)), algfunc, y );   yields false
```

SEE ALSO : type radical

9.1.119.2. type/algnum – check for an algebraic number

CALLING SEQUENCE : type(a,algnum);
PARAMETERS : a – any expression

- The call type(a,algnum) checks to see if "a" is an algebraic number. An algebraic number is defined as either a rational number, or a root of a univariate polynomial with algebraic number coefficients, specified by a RootOf. For example, the algebraic number 2^(1/2) could be specified as RootOf(z^2-2,z). A sum, product or quotient of these is also an algebraic number.

EXAMPLES:

```
type(2/3,algnum);                  yields  true
type(ln(2),algnum);                yields  false
type(RootOf(z^2+1,z),algnum);      yields  true
type(5/RootOf(z^2-2,z),algnum);    yields  true
type(RootOf(z^2+y,z),algnum);      yields  false
```

SEE ALSO : RootOf, help(convert,RootOf)

9.1.119.3. type/boolean, type/relation and type/logical

CALLING SEQUENCES : type(x,boolean); type(x,relation); type(x,logical)
PARAMETER : x – any expression

- A relational expression is an expression of type `=`, `<>`, `<` or `<=`.
- A logical expression is an expression of type `and`, `or`, or `not`.
- A boolean expression is an expression of type relation or type logical or one of the boolean constants "true" or "false".

9.1.119.4. type/constant – check for constant

CALLING SEQUENCE : type(x,constant);
type(x,constant,complex);
type(x,constant,real);

PARAMETERS : x – any expression

- The call type(x,constant) or type(x,constant,complex) checks to see if "x" is a complex constant.
- The call type(x,constant,real) checks if "x" is a real constant. A real constant is infinity, –infinity, or an expression for which evalf will return a floating point number.

EXAMPLES:

type(ln(Pi),constant,real);	yields	true
type(ln(–Pi),constant,real);	yields	false
type(ln(–Pi),constant);	yields	true
type(infinity,constant);	yields	true
type(x^2,constant);	yields	false

SEE ALSO : indets

9.1.119.5. type/expanded – check for expanded polynomial

CALLING SEQUENCE : type(<expr>, expanded);
PARAMETERS : <expr> – a polynomial

- Assuming that type(<expr>, polynom) is true, then
  type(<expr>, expanded)
  will return true if <expr> is an expanded polynomial, false otherwise.

EXAMPLES:

type((4*x*y–x)*(x+y), expanded);	yields	false
type(4*x^2*y + 4*x*y^2 – x^2 – x*y, expanded);	yields	true

SEE ALSO : type, help(type,polynom)

9.1.119.6. type/laurent – check for laurent series

CALLING SEQUENCE : type(<expression>, laurent);

- Type/laurent returns true if <expression> is of type series and only contains power terms of the variable in which it was expanded.

EXAMPLES:

```
taylor(sin(x),x,5);      yields x-1/6*x^3+O(x^5)
type(",laurent);         yields true
taylor(1/sin(x),x,5)     yields x^(-1)+1/6*x+O(x^3)
type(",laurent);         yields true
taylor(ln(x+x^2),x,3);   yields ln(x)+x-1/2*x^2+O(x^3)
type(",laurent);         yields false
```

SEE ALSO : series, taylor, type, help(type,taylor)

9.1.119.7. type/linear, type/quadratic, type/cubic, type/quartic

CALLING SEQUENCES : type(a,linear,v); and type(a,quadratic,v);
type(a,cubic,v); and type(a,quartic,v);

PARAMETERS : a – an expression
v – an indeterminate or a list or set of indeterminates

- Check if the expression "a" is linear (quadratic, cubic or quartic) in the indeterminates "v". If only two arguments are specified, this is equivalent to the call type(a,linear,indets(a));. That is, "a" must be linear (quadratic, cubic, quartic) in all of its indeterminates.
- The definition of linear in the indeterminates "v" is:
  type(a,polynom,v) and degree(a,v) = 1
  where "degree" means "total degree" in the case of several variables. The definitions for quadratic, cubic and quartic are analogous with degree(a,v) = 2, 3 and 4 respectively.

EXAMPLES:

```
type(x*y+z, linear, x);              yields    true
type(x*y+z, linear);                 yields    false
type(x*y+z, quadratic);              yields true
type(x^2+y, linear);                 yields    false
type(f(1)*x+2^(1/2), linear);        yields    true
type(x/z+y/g(2), linear, [x,y]);     yields    true
```

SEE ALSO : indets, degree, help(type,polynom)

9.1.119.8. type/listlist – check for a list of lists

CALLING SEQUENCE : type(e, listlist)

- "type/listlist" returns true if "e" is a list of lists, with each inner list having the same number of elements. Otherwise, it returns false.

EXAMPLES:

```
type( [[1,2],[3,4]], listlist );          yields  true
type( [8,7,6,12], listlist );             yields  false
type( [[1,2],[3,4,5]], listlist );        yields  false
type( [[]], listlist );                   yields  true
type( [ [[a],[b]], [[c],[d]] ], listlist ); yields  true
```

SEE ALSO: help(convert,listlist)

9.1.119.9. type/matrix – check for matrix (two-dimensional array)

CALLING SEQUENCE : type(A,matrix); or type(A,matrix,square);
PARAMETERS : A – maple expression

- Check for a Maple matrix. A matrix is defined to be a two-dimensional array with indices starting at 1. If a second argument is given, and it is the name "square", type matrix also checks whether the upper indices are equal.

EXAMPLES:

```
type(array(1..2,1..2),matrix);  yields  true
type(array(0..2,0..2),matrix);  yields  false
type(array(1..2,1..2),matrix,square);  yields  true
type(array(1..2,1..4),matrix,square);  yields  false
```

9.1.119.10. type/monomial – check for monomial

CALLING SEQUENCE : type(m,monomial); or type(m,monomial,v);
PARAMETERS : m – any expression
v – (optional) variable(s)

- The call type(m, monomial, v) checks to see if "m" is a monomial in the variable(s) "v" where "v" is an indeterminate, or a list or set of indeterminates. A monomial is defined to be a polynomial in "v" which does not contain a sum in it.
- If only two arguments are specified, this is equivalent to calling type(m, monomial, indets(m)). That is, "m" must be a polynomial in all of its indeterminates.

EXAMPLES:

```
type(sin(1)*x^2,monomial);      yields  true
type(sin(1)/x^2,monomial);      yields  false
type((1+y)*x^2,monomial,y);     yields  false
type(sin(x)*y,monomial,y);      yields  true
```

SEE ALSO : indets, type, help(type,polynom)

9.1.119.11. type/point – check for a point

CALLING SEQUENCE : type(P,point)
PARAMETERS : P – any algebraic expression

- Return true if the object is an equation with only a name on the left hand side and an algebraic expression on the right hand side. Return false otherwise.
- If the object is a set, return true if all members of the set are of type point, false otherwise.
- The left-hand-side names are intended to define the coordinate space, and the right-hand-side values are the corresponding coordinate values.
- Objects of type `point` differ from objects of type `Point` (see the student package) in that the coordinates are defined by names rather than position in a list.

EXAMPLES :

```
type( a=4, point );              yields true
type( {a=4,b=2}, point );        yields true
type( {a+b=c,d=3} , point );     yields false
```

SEE ALSO : type, limit, help(student,Point)

9.1.119.12. type/polynom – check for polynomial

CALLING SEQUENCE : type(a,polynom,v); or type(a,polynom,v,d);
PARAMETERS : a – any expression
v – variable(s)
d – type name for the coefficient domain

- The call type(a,polynom,v,d) checks to see if "a" is a polynomial in the variables "v" with coefficients in the domain "d". A typical calling sequence would be type(a,polynom,x,integer) which tests to see if "a" is a polynomial in "x" over the integers.
- The variable(s) "v" can be a single indeterminate, or, it can be a list or set of indeterminates in which case one would be testing for a multivariate polynomial.
- The domain specification "d" is a type name, such as rational, or algnum (algebraic number). If the domain specification is omitted, then no restriction is placed on the coefficients.
- If the variables are omitted, then "d" must be one of integer, rational, numeric or constant. This special case tests for a polynomial in names over "d".
- If the variables and the domain specification are omitted, this special case is equivalent to type(a,polynom,constant).

EXAMPLES:

type(x^2+y–z,polynom);	yields	true
type(sin(x)+y,polynom,x);	yields	false
type(sin(x)+y,polynom,y);	yields	true
type(f(1)*x+2^(1/2),polynom);	yields	true
type(x^2+y^3/3,polynom,[x,y]);	yields	true
type(x+1/2,polynom,integer);	yields	false
type(x+1/2,polynom,rational);	yields	true

SEE ALSO : indets, type, help(type,ratpoly), help(type,monomial)

9.1.119.13. type/radical and type/sqrt – check for fractional powers

CALLING SEQUENCES : type(e,radical); or type(e,radical,v)
type(e,sqrt); or type(e,sqrt,v);
PARAMETERS : e – an expression
v – variable(s)

- The definition of type "radical" is that "e" is of type `^` and op(2,e) (the exponent) is of type "fraction".

- If an additional argument "v" is provided, where "v" is an indeterminate or a list or set of indeterminates, op(1,e) (the base) must contain v, that is, has(op(1,e),v) .
- An expression is of type sqrt if it is of type radical and the denominator of op(2,e) (the exponent) is 2 .

EXAMPLES:

```
type( 2^(1/2), radical );      yields true
type( 2^(1/2), sqrt );         yields true
type( y^(2/5), radical );      yields true
type( y^(2/5), sqrt );         yields false
type( y^3, radical );          yields false
type( y^(2/5), radical, y );   yields true
type( y^(2/5), radical, x );   yields false
```

SEE ALSO : `^`, fraction, has

9.1.119.14. type/ratpoly – check for rational function

CALLING SEQUENCE : type(a,ratpoly,v); or type(a,ratpoly,v,d);

PARAMETERS : a – any expression
v – variable(s)
d – type name for the coefficient domain

- The call type(a,ratpoly,v,d) checks to see if "a" is a rational function in the variables "v" with coefficients in the domain "d". A typical calling sequence would be type(a,ratpoly,x,integer) which tests to see if "a" is a rational function in "x" over the integers.
- The variable(s) "v" can be a single indeterminate, or, it can be a list or set of indeterminates.
- The domain specification "d" is a type name such as integer or algnum (algebraic number). If the domain specification is omitted, then no restriction is placed on the coefficients.
- If the variables are omitted, then "d" must be one of integer, rational, numeric or constant. This special case tests for a rational function in names over "d".
- If both the variables and the domain specification are omitted, then this is equivalent to type(a,ratpoly,constant).

EXAMPLES:

```
type((1+x)/(1-y),ratpoly);          yields  true
type((1+sin(x))/y,ratpoly);         yields  false
type(1/f(1)+1/x,ratpoly);           yields  true
type(sin(z)-cos(z)/x,ratpoly,x);    yields  true
type(x+sqrt(1-x),ratpoly,x);        yields  false
```

SEE ALSO : indets, help(type,polynom);

9.1.119.15. type/scalar – check for scalar (in the matrix sense)

CALLING SEQUENCE : type(<expr>, scalar);
PARAMETERS : <expr> – maple expression

- Checks if <expr> is a scalar – that is, not a vector or matrix.

EXAMPLES:

```
a := array([[1,2],[3,4]]):
type(a,scalar);                yields  false
type(det(a),scalar);           yields  true
type(x,scalar);                yields  true
type(42,scalar);               yields  true
```

9.1.119.16. type/taylor – check for taylor series

CALLING SEQUENCE : type(<expression>, taylor);

- Type/taylor returns true if <expression> is a series which is a polynomial in the variable in which it was expanded. An order term may be present.

EXAMPLES:

```
taylor(sin(x),x,5);     yields  x-1/6*x^3+O(x^5)
type(",taylor);         yields  true
taylor(1/sin(x),x,5)    yields  x^(-1)+1/6*x+O(x^3)
type(",taylor);         yields  false
type(x^3,taylor);       yields  false
taylor(x/y,x);          yields  1/y*x
type(",taylor);         yields  true
taylor(x/y,y);          yields  x*y^(-1)
type(",taylor);         yields  false
```

SEE ALSO : series, taylor, type, help(type,laurent)

9.1.119.17. type/trig – check for trigonometric functions

CALLING SEQUENCE : type(e,trig,x);
type(e,trig);
PARAMETERS : e – any expression
x – a variable name

- The call type(e,trig) returns true if "e" is a function and the function name is one of the trigonometric functions:
sin, cos, tan, sec, csc, cot, sinh, cosh, tanh, sech, csch, coth
- The call type(e,trig,x) checks, in addition, that the function e contains the variable name x.

EXAMPLES:

type(sin(x), trig);	yields	true
type(exp(x), trig);	yields	false
type(sin(x) + cos(x), trig);	yields	false
type(sin(1), trig, x);	yields	false
type(tanh(3*x–1), trig, x);	yields	true

9.1.119.18. type/vector – check for vector (one-dimensional array)

CALLING SEQUENCE : type(<expr>, vector);
PARAMETERS : <expr> – maple expression

- Checks if <expr> is a maple vector. A vector is defined to be a one dimensional array indexed from 1.

EXAMPLES:

type(array(1..3,[x,y,z]),vector);	yields	true
type(array(0..2,[x,y,z]),vector);	yields	false

9.1.120. unames – sequence of unassigned names

CALLING SEQUENCE : unames( );

- "unames" returns an expression sequence consisting of all the active names in the current Maple session which are "unassigned names".
- An "unassigned name" is a name (or string) that has been used in the current Maple session but has no value other than its own name.
- Note, since file names and error messages are strings, these are also unassigned names.

SEE ALSO : anames

9.1.121. whattype – query the basic data type of an expression

CALLING SEQUENCE : whattype(<expr>);
PARAMETER : <expr> – any expression

- "whattype" returns the data type name of <expr>, which may be any of the following basic data types: `+`, `*`, `^`, `=`, `<>`, `<`, `<=`, `.`, `..`, `and`, `or`, `not`, exprseq, float, fraction, function, indexed, integer, list, procedure, series, set, string, table, uneval.
- Although "exprseq" is not a type name known to the "type" function, it is the name of the internal data structure for expression sequences.
- For a general expression, "whattype" returns the "top level" data type as determined by the order of precedence of the operators.

EXAMPLES :

```
whattype( x + y );      yields  +
whattype( x – y );      yields  +
whattype( –x );         yields  *
whattype( x^2*f(y) );   yields  *
whattype( x/y );        yields  *
whattype( x^y );        yields  ^
whattype( 1/x );        yields  ^
whattype( x, y );       yields  exprseq
```

SEE ALSO : type, precedence, help(index, datatypes)

9.1.122. with – define the names of functions from a library package

CALLING SEQUENCE : with(<package>); or with(<package>, f1, f2, ...);
PARAMETERS : <package> – name of a Maple package
f1, ... – (optional) names of functions

- The effect of the "with" function is to define a function name f (or a group of function names) to point to the same procedure as the corresponding package name package[f].
- If only one argument is supplied to "with" then all the functions in the package will be defined. If additional arguments are specified then they are taken to be the names of functions in the package and only these specific functions will be defined.
- The value returned is a list of the functions defined. A warning message is printed for each name that had a previous value.
- For example, the result of invoking with(linalg); is to define a group of names (add, inverse, etc.) to point to the same procedures as their corresponding package names linalg[add], linalg[inverse], etc.

EXAMPLE :

```
A := array([7]);        yields  A := array( 1..1, [(1)=7] )
with( linalg, add );    yields  [add]
add( A, A );            yields  array( 1..1, [(1)=14] )
```

SEE ALSO : help(index, packages);

9.1.123. words – query memory usage (words used)

CALLING SEQUENCE : words(n);
PARAMETERS : n – an optional integer

- The function "words" returns the total words used. The size of a word depends on the computer hardware and is usually 32 or 36 bits. This measure is a cumulative count of all memory requests made up to that point to the Maple storage manager, which typically includes some reused memory.
- During a Maple session, a message of the form:
  words used=xxxx, alloc=yyyy, time=zzzz
  is printed out at the first "safe point" after every 100,000 words used (or some other system-specific default value). Here xxxx is the total words used, yyyy is the total words actually allocated, and zzzz is the total CPU time used (in seconds) since the start of the session.
- The integer parameter "n" (if present) controls the frequency of the "words used ..." message. If "n" is less than or equal to zero then the "words used ..." message will not be printed (except when a garbage collection occurs). The current value of this frequency is available as status[4], the fourth element of the 'status' sequence.

SEE ALSO : 'status', gc

9.1.124. writeto – write output to a file

CALLING SEQUENCE : writeto(<filename>);
PARAMETER : <filename> – a name

- When the command "writeto(<filename>);" is typed all future commands will have their results immediately stored in <filename> and will not be displayed on the screen.
- While in this mode, no prompt will appear though the commands typed will be displayed.
- The special command "writeto(terminal);" will restore the standard mode of printing the results on the terminal.

- If <filename> exists the contents of the file are overwritten.

SEE ALSO : `save`, write

9.1.125. Zeta – the Riemann zeta function

The Riemann zeta function is defined by

$$\text{Zeta}(s) = \sum_{i=1}^{\infty} \frac{1}{i^s}, \quad \text{for } s > 1$$

and its derivatives are represented by

$$\text{Zeta}(n, s) = \text{diff}(\text{Zeta}(s), s\$n), \quad \text{for } n \geq 1 \ .$$

Automatic simplifications will be applied to these functions.

9.2. Miscellaneous library functions

9.2.1. bspline – compute the B-spline segment polynomials

CALLING SEQUENCE : bspline(d,v); or bspline(d,v,k);
PARAMETERS : d – non-negative integer
v – name
k – (optional) list of d+2 numbers or symbols

- The bspline function must first be loaded by "readlib(bspline);" . It computes the segment polynomials for the B-spline of degree "d" in the symbol "v" on the knot sequence "k" . If "k" is not specified, the uniform knot sequence [0,1,...,d+1] is used.
- If the knot sequence contains symbols, the symbolic representation of the B-spline segment polynomials is returned. Additionally, the knot sequence can have multiple knots. Multiple knots result in a loss of continuity at the knot. If the multiplicity of a knot is 1 <= m <= d, then the continuity at that knot is d–m .
- For numerical knots, the B-spline can be plotted by doing b := convert(bspline(d,v),procedure); plot(b,v=0..d+1); Note that this results in a temporary file called "temp" being created.

EXAMPLES :
```
readlib(bspline):
bspline(1,u); yields
   [[u<0,0], [u<1,u], [u<2,–u+2], [2<=u,0]]
bspline(1,u,[0,a,2]); yields
   [[u<0,0], [u<a,u/a], [u<2,(–u+a)/(2–a)+1], [2<=u,0]]
```

9.2.2. C – the Fresnel cosine integral

The Fresnel cosine integral is defined by

$$C(x) = \int_0^x \cos(\frac{\pi}{2}t^2)\ dt\ .$$

If this function is read in via "readlib(C):" then simplifications will be applied.

9.2.3. chrem – Chinese Remainder Algorithm

CALLING SEQUENCE : chrem(u, m)

PARAMETERS : u – the list [u0,u1,...,un] of evaluations
m – the list of moduli [m0,m1,...,mn]

- If the u.i's are integers, chrem(u,m) computes the unique positive integer a such that a mod m0 = u0, a mod m1 = u1, ..., a mod mn = um, and a < m0 m1 ... mn . The moduli m0,m1,...,mn must be pairwise relatively prime.
- If the u.i's are polynomials, chrem is applied to the coefficients of each term in {u0,u1,...,um} .
- This function must be loaded via: readlib(chrem):

EXAMPLES :
readlib(chrem):
chrem([1,2], [5,7]); yields 16
chrem([3*x,2*y+x], [5,7]); yields 8*x + 30*y

9.2.4. Ci – the cosine integral

The cosine integral is defined by

$$\mathrm{Ci}(x) = \mathrm{gamma} + \ln(x) + \int_0^x \frac{\cos(t) - 1}{t}\, dt \ .$$

If this function is read in via "readlib(Ci):" then simplifications will be applied.

9.2.5. coeftayl – coefficient of (multivariate) expression

CALLING SEQUENCE : coeftayl(expr, eqn, k)
PARAMETERS : expr – an arbitrary expression
eqn – an equation of the form x = alpha where x is a name (univariate case) or list (multivariate case)
k – a nonnegative integer (univariate case) or a list of nonnegative integers (multivariate case)

- This function must first be loaded via: readlib(coeftayl):
- This function computes a coefficient in the (multivariate) Taylor series representation of expr without forming the series (it uses differentiation and substitution). Often, expr is a polynomial.
- The one-variable and several-variable cases are distinguished by the types of the input parameters.

- *UNIVARIATE CASE*: x is a name and k a nonnegative integer.
- In this case, the value returned is the coefficient of (x–alpha)^k in the Taylor series expansion of expr about x = alpha. This is equivalent to executing coeff(taylor(expr, x=alpha, k+1), x–alpha, k) but it is more efficient (because only a single term is computed).
- *MULTIVARIATE CASE*: x is a nonempty list [x1,...,xv] of indeterminates appearing in expr and alpha is a list [alpha1,...,alphav] specifying the point of expansion with respect to the given indeterminates; k is a list [k1,...,kv] of nonnegative integers corresponding to elements in x and alpha.
- In this case, the value returned is the coefficient of the term specified by the monomial
  (x[1]–alpha[1])^k[1] * . . . * (x[v]–alpha[v])^k[v]
  in the multivariate Taylor series expansion of expr about the "point" x = alpha. If k is the list of zeros then the value returned is the value resulting from substituting x = alpha into expr.

9.2.6. commutat – commutator package

c, expand/c, convert/c, simplify/c, &*, expand/&*, convert/&*

Definition : the commutator c(x,y) = x y – y x

where multiplication here is non-commutative. This package provides for the manipulation and simplification of commutators, expanding commutators in terms of &* (Maple's non-commutative multiplication operator), and converting an expression in terms of &* to commutator form. The following identities are applied automatically:

c(y,x) = – c(x,y)
c(x,x) = 0 for all x
c(k,x) = 0 for any constant k
c(x+y,z) = c(x,y) + c(y,z)
c(k*x,y) = k*c(x,y) for any constant k .

A canonical form for commutators is obtained by applying the following two rules given the ordering: order by number of nested commutators, and break ties by address of object.

c(y,x) ==> –c(x,y) iff x < y
c(x,c(y,z)) ==> c(y,c(x,z)) – c(z,c(x,y)) iff x < y .

In particular, this form recognizes the Jacobi identity:

c(x,c(y,z)) + c(y,c(z,x)) + c(z,c(x,y)) = 0 .

Commutators can be expressed in terms of &* as in

convert(c(x,(y,z)), `&*`);

```
– &*(y, z, x) – &*(x, z, y) + &*(z, y, x) + &*(x, y, z)
```

Or converted back in terms of c

```
convert(",c);
                    c(y, c(x, z)) – c(z, c(x, y))
```

N.B. If the expression being converted to a commutator cannot be expressed as a commutator, the result returned will be pure nonsense. The utility routine commutat can be used to display commutators as a list of lists syntax.
This package must first be loaded via readlib(commutat):

9.2.7. compoly – determine a possible composition of a polynomial

CALLING SEQUENCE : compoly(r, x);
PARAMETER : r – a polynomial
x – a variable on which the composition will be made

- compoly returns a pair p(x), x=q(x) such that subs(x=q(x), p(x)) is equal to ``r", the input polynomial. If such a pair cannot be found, it returns FAIL. p(x) and q(x) are non-linear polynomials and q(x) has a low-degree in ``x" greater or equal to 1.
- Note that the composition may not be unique.
- This function has to be loaded through: readlib(compoly):

EXAMPLES :

```
compoly( x^6–9*x^5+27*x^4–27*x^3–2*x^2*y+6*x*y+1, x );
        yields  x^3–2*x*y+1, x = x^2–3*x;
compoly( x^4–3*x^3–x+5, x );
        yields  FAIL
```

9.2.8. convergs – print convergents of continued fraction

CALLING SEQUENCE : convergs(<listA>, <listB>, n);
PARAMETERS : <listA> – list of numbers, or a function
<listB> – (optional) list of numbers
n – (optional) integer

- "convergs" takes the continued fraction:
  a1 + b2/(a2 + b3/(a3 + b4/(a4 + . . .
  and prints the successive convergents in the form: n, nth convergent. "convergs" returns NULL.
- <listA> is of the form: [a1, a2, a3, ...] and similarly <listB> is of the form: [1, b2, b3, ...]; where "aI" and "bI" correspond the values in the displayed continued fraction.

- If <listB> contains is of the form: [1, 1, 1, ...] it may be omitted.
- If a third argument, "n", is present it indicates the number of convergents to compute.
- "convergs" must first be read in by the statement: "readlib(convergs):"

SEE ALSO : convert, help(convert,confrac)

9.2.9. D – Differential operator

CALLING SEQUENCE : D(f)
PARAMETER : f – expression which can be applied as a function

- D(f) is a function defined so that D(f)(x) = diff(f(x),x). That is, D(f) = unapply(diff(f(x),x),x).
- The argument f must be an algebraic expression which can be treated as if it were a function. All names are treated as function names. For example, f= g+h and f = f*h are valid since (g+h)(x) = g(x) + h(x) and (g*h)(x) = g(x)*h(x).
- The usual rules for differentiation hold including, for example, D(sin) = cos and D(1/f) = –D(f)/f^2 . Functional composition is represented by g @ h.
- This function has to be loaded with readlib(D):.

EXAMPLES :
D(f) yields D(f)
D(f*g); yields D(f) g + f D(g)
D(f@g); yields (D(f) @ g) D(g)
D(y^2=x^3 +x); yields 2 D(y) y = 3 D(x) x^2 + D(x)
SEE ALSO : diff, `@`, `@@`, unapply, help(operators,example), help(operators,D).

9.2.10. dilog – the dilogarithm integral

The dilogarithm integral is defined by

$$\text{dilog}(x) = \int_1^x \frac{\ln(t)}{1-t}\,dt \ .$$

If this function is read in via "readlib(dilog):" then simplifications will be applied.

9.2.11. dinterp – probabalistic degree interpolation

CALLING SEQUENCE : dinterp(f,n,k,d,p)
PARAMETERS : f – a Maple procedure
n,k,d – integers
p – prime modulus

- Given a function "f" that evaluates a polynomial in "n" variables modulo "p", and a degree bound "d" on the k'th variable, determine probabilistically the degree of the k'th variable.
- Dinterp may return FAIL if it encounters a division by zero when evaluating "f". It may also return a result for the degree of the k'th variable which is too low. The probability that this happens can be decreased by using a larger modulus. A 12 to 20 digit modulus is considered ideal.
- This function has to be loaded with the command: readlib(dinterp):

EXAMPLES :

```
f := proc(x,y,z,p) x^2+y^3+z^4 mod p end:
dinterp(f,3,1,6,997); yields 2
dinterp(f,3,2,6,997); yields 3
dinterp(f,3,3,6,997); yields 4
```

9.2.12. discrim – compute the discriminant of a polynomial

CALLING SEQUENCE : discrim(p, x)
PARAMETERS : p – polynomial in x
x – independent variable

- If d=degree(f,x) and a=lcoeff(f,x) then the discriminant is (–1)^(d*(d–1)/2)*resultant(p,diff(p,x),x)/a
- This function must first be loaded via readlib(discrim):

EXAMPLES :

```
p := a*x^2 + b*x + c;
readlib(discrim):
discrim(p,x); yields –4*a*c + b^2
```

9.2.13. edit – expression editor

CALLING SEQUENCE : edit(<expr>);
PARAMETERS : <expr> – any expression

- This expression editor is invoked as: edit(<expr>); The <expr> is then displayed with a "BOX" around it. The "BOX" is moved around within <expr> by commands. The name _ is assigned the contents of the "BOX".
- This function must be loaded via readlib(edit):

COMMANDS :

up;	move BOX to enclosing expression
down;	move BOX to 1st operand of expression
top;	move BOX to include the whole expression
succ;	move BOX to successive subexpr at current level
prev;	move BOX to previous subexpr at current level
first;	move BOX to first subexpr at current level
last;	move BOX to last subexpr at current level
child(4,1,2);	move BOX to op(2,op(1,op(4,the expression)))
find(XXX);	move BOX to the next occurrence of XXX
use(YYY);	YYY replaces the BOX's contents. (YYY is usually some function of _)
show;	display the expression

EXAMPLES :

edit(x^2+x^3);	yields	BOX(x^2+x^3)
down;	yields	BOX(x^2)+x^3
down;	yields	BOX(x)^2+x^3
use(sin(_));	yields	BOX(sin(x))^2+x^3
top;	yields	BOX(sin(x)^2+x^3)
_;	yields	sin(x)^2+x^3

9.2.14. Ei – the exponential integral

The exponential integral is defined by

$$\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{\exp(t)}{t}\, dt \ .$$

If this function is read in via "readlib(Ei):" then simplifications will be applied.

9.2.15. eqn – produce output suitable for troff/eqn printing

CALLING SEQUENCE : eqn(expr, filename)
PARAMETERS : expr – any expression
filename – (optional) output to a file

- ``eqn" produces output which is suitable for printing ``expr" with a troff/eqn processor.
- The format of the output depends on the degree of simplification of expr, no attempt is made to modify the input expression. The mathematical format is taken, in general, from the CRC handbook or the Handbook of Mathematical functions.
- The functions sum, int, diff, limit and log are aliased by Sum, Int, Diff, Limit and Log, so that these can be used to prevent evaluation by maple.
- User-defined function printing can be interfaced by including a procedure `eqn/<function-name>`.
- ``eqn" will not produce the ".EQ" or ".EN" lines in the output file. (Some peculiarities of the UofW typesetter are included).
- This function has to be loaded via readlib(eqn):

EXAMPLES :

```
writeto(TroffFile);
eqn( Int( 1/(x^2+1), x) = int( 1/(x^2+1), x) );
writeto(terminal);
eqn( x^2 + y^2 = z^2, OtherTroffFile );
```

9.2.16. eulermac – Euler-Maclaurin summation

CALLING SEQUENCE : eulermac(<expr>, x, n)
PARAMETERS : <expr> – an expression in "x"
x – independent variable
n – integer (degree of summation)

- This function must first be loaded via: readlib(eulermac):
- "eulermac" computes an n'th degree Euler-Maclaurin summation formula of <expr>.
- "eulermac" is an asymptotic approximation of sum(<expr>, x). If F(x) := eulermac(f(x), x), then F(x+1) – F(x) is asymptotically equivalent to f(x).

9.2.17. evalm – matrix evaluation

CALLING SEQUENCE : evalm(<matrix expression>)
PARAMETER : <matrix expression> – an expression

- evalm evaluates an expression involving matrices. It performs any sums, products, or integer powers involving matrices, and will map functions onto matrices.
- This function must be read in using: readlib(evalm):
- Note that maple may perform simplifications before passing the arguments to evalm, and these simplifications may not be valid for matrices. For example, evalm(a*b) and evalm(b*a) will produce the same result, since maple's multiplication operator assumes commutativity. Also, evalm(a^0) will return 1, not the identity matrix.
- To indicate non-commutative matrix multiplication, use the operator &*. Note that an &-operator in Maple must be separated from its operands by at least one blank.

EXAMPLES :

```
readlib(evalm):
s := array([[1,2],[3,4]]):
t := array([[1,1],[2,-1]]):
evalm(s+2*t);          yields array(1..2,1..2,[(1,1)=3,(1,2)=4,
                       (2,1)=7,(2,2)=2])
evalm(s^2);            yields array(1..2,1..2,[(1,1)=7,(1,2)=10,
                       (2,1)=15,(2,2)=22])
evalm(sin(s));         yields array(1..2,1..2,[(1,1)=sin(1),
                       (1,2)=sin(2),(2,1)=sin(3),(2,2)=sin(4)])
evalm(s &* t);         yields array(1..2,1..2,[(1,1)=5,(1,2)=-1,
                       (1,3)=11,(1,4)=-1])
evalm(s * t);          may yield the same as above, or it may yield
evalm(t &* s);         which is array(1..2,1..2,[(1,1)=4,(1,2)=6,
                       (2,1)=-1,(2,2)=0])
```

SEE ALSO : array, linalg

9.2.18. fibonacci – compute Fibonacci numbers

CALLING SEQUENCE : fibonacci(n);
PARAMETERS : n – any expression

- The procedure "fibonacci" computes the n'th Fibonacci number, if the argument n is an integer, otherwise it returns the unevaluated function call.

- The Fibonacci numbers are defined by: fibonacci(0)=0, fibonacci(1)=1 and fibonacci(n) = fibonacci(n–1) + fibonacci(n–2) .
- This function must be loaded before using it. It may be loaded via: "readlib(fibonacci):" or, since it is part of the numtheory package, you may use: with(numtheory);

EXAMPLES :

fibonacci(6); yields 8
fibonacci(n); yields fibonacci(n)

9.2.19. fixdiv – compute the fixed divisor of a polynomial

CALLING SEQUENCE : fixdiv(a, x);
PARAMETERS : a, x – a is polynomial in x over the integers

- Compute the fixed divisor of a(x) in Z[x]. The fixed divisor is the largest integer that divides a(k) for all k in Z . For primitive polynomials, the fixed divisor is no greater than n! where n = degree(a,x) .
- It is known that the probability that the fixed divisor is greater than 1 for random polynomials of degree n, is approximately 0.278 as n goes to infinity.
- This function must be loaded via: readlib(fixdiv):

9.2.20. forget – remove an entry or entries from a remember table

CALLING SEQUENCE : forget(f, a,b,c,...); or forget(f);
PARAMETERS : f – any name assigned to a maple procedure
a,b,c,.. – specific argument sequence for the function f

- The statement "forget(f);" will remove all entries from the remember table of f, while "forget(f,a,b,c);" will remove only the entry corresponding to f(a,b,c).
- Maple procedures with option remember automatically remember every value computed by inserting an entry in the remember table.
- Users may assign f(a,b,c) a specific value as in "f(a,b,c) := 3;". This assigned value is kept in the remember table of the procedure f. Once assigned a value in this manner any future reference to f(a,b,c) will return "3", without recomputing the function value. To allow f to recompute its function value, assigned values must be removed from the remember table.
- You need not use forget if your goal is simply to reassign f(a,b,c) a different value. For example, just use "f(a,b,c) := 5;".

- This function must be loaded via readlib(forget):

EXAMPLES :
```
readlib(forget):
f(x),f(y);    yields    f(x) , f(y)
f(x) := 456:
f(x),f(y);    yields    456 , f(y)
forget(f,x);
f(x),f(y);    yields    f(x) , f(y)
```
SEE ALSO : `options`, remember, procedures

9.2.21. fortran – generate fortran source output

CALLING SEQUENCE : fortran(s, ...);
PARAMETERS : s an expression or equation or list of equations

- This function must first be loaded by typing readlib(fortran):
- fortran generates fortran source output corresponding to the statements in s. Currently, fortran cannot generate output for Maple procedures. s must be an expression or equation (assignment) or a list of equations corresponding to a sequence of assignments.
- By default the output stream is standard output but an optional argument of the form filename = "f" can be used to direct the output to the file "f". If the keyword "optimized" is specified, common subexpression optimization is performed. The global variable precision can be assigned either single or double (single by default) for single or double precision respectively.

EXAMPLES :
```
readlib(fortran):
s := ln(x)+2*ln(x)^2-ln(x)^3:
fortran(s);                yields   alog(x)+2*alog(x)**2-alog(x)**3
fortran(s,optimized);      yields   t1 = alog(x)
                                    t2 = t1**2
                                    t6 = t1+2*t2-t1*t2
```
SEE ALSO : optimize

9.2.22. freeze, thaw – replace an expression by a name

CALLING SEQUENCE : freeze(<expr>); thaw(<var>);
PARAMETERS : <expr> – the expression to be "frozen"
<var> – the "frozen" variable

- "freeze" replaces its argument, <expr> (not a name and not a rational number), with a name of the form _R0, _R1, ...
- To return the original expression the thaw function must be used.
- The functions "freeze" and "thaw" must first be loaded via: readlib(freeze):

EXAMPLE :

```
readlib(freeze):
freeze(x+y);    yields  _R0;
thaw(_R0);      yields  x+y;
```

9.2.23. hypergeom – generalized hypergeometric function

CALLING SEQUENCE : hypergeom([n1,n2,...], [d1,d2,...], z)
PARAMETERS : [n1,n2,...] – list of numerator coefficients
[d1,d2,...] – list of denominator coefficients

- The function hypergeom(n,d,z) is the generalized hypergeometric function F(n,d,z). This function is also known as Barnes's extended hypergeometric function. If there are j coefficients in n, and k coefficients in d, this function is also known as jFk.
- The definition of F(n,d,z) is sum ((product(GAMMA(n[i]+k), i=1..j)*z^k) / (product(GAMMA(d[i]+k), i=1..m)*k!), k=0..infinity), where j is the number of terms in the list [n1, n2, ...] and m is the number of terms in the list [d1, d2, ...].
- If n[i] = –m, a nonpositive integer, the series stops after m terms.
- This routine must be loaded before use with "readlib(hypergeom):".

EXAMPLES :

```
readlib(hypergeom):
hypergeom( [],[],z );          yields  exp(z)
hypergeom( [a],[],z );         yields  (–1)^(–a)*(z–1)^(–a)
hypergeom( [1,2],[2,3],z );    yields  hypergeom([1],[3],z)
```

9.2.24. invlaplace – inverse Laplace transform

CALLING SEQUENCE : invlaplace(expr, s, t);
PARAMETERS :
- expr – expression to be transformed
- s – variable "expr" is transformed with respect to
- t – variable to replace "s" in transformed expression

- "invlaplace" applies the inverse Laplace transformation to "expr" with respect to "s".

- Expressions which are sums of rational functions of polynomials can be transformed.
- "invlaplace" and "laplace" recognize the Dirac-delta (or unit-impulse) function as Dirac(t).
- This function must be loaded by entering "readlib(laplace):".

EXAMPLES :
```
readlib(laplace):
invlaplace(1/(s-a)+1/(s^2-b^2)+1,s,t);     yields
                    exp(a*t) + sinh(b*t)/b + Dirac(t)
invlaplace(1/(s^3-8)+laplace(y(t),t,s),s,t);     yields
     1/12*exp(2*t) - 1/4*exp(-t)*sin(sqrt(3)*t)/sqrt(3) -
                                1/12*exp(-t)*cos(sqrt(3)*t) + y(t)
```
SEE ALSO : laplace

9.2.25. iroot – integer nth root

CALLING SEQUENCES : isqrt(x); and iroot(x,n);
PARAMETERS : x, n – integers

- "isqrt" computes an integer approximation to the square root of x. The approximation is exact for perfect squares, and the error is less than 1 otherwise. Note: if $x < 0$, isqrt(x) returns 0.
- "iroot" computes an integer approximation to the n'th root of x. The approximation is exact for perfect powers, and the error is less than 1 otherwise. Note: if $x < 0$, and n is even, iroot(x,n) returns 0.
- The iroot function must first be read in via "readlib(iroot):"

9.2.26. iscont – test continuity on an interval

CALLING SEQUENCE : iscont(expr, x = a .. b);
PARAMETERS :
	expr	– an expression
	x	– variable
	a .. b	– a real interval

- This function must be read into a Maple session by typing: "readlib(iscont):".
- "iscont" returns true if the expression is continuous on the interval (that is, it has no poles), or false if the expression is not continuous. If "iscont" can't determine the result it returns FAIL.

- "iscont" assumes that any symbols in the expression are real.
- The interval is considered to be an open interval. That is, the endpoints are excluded from the range.
- The endpoints of the interval must be either real constants or "infinity" or "–infinity".
- The endpoints of the interval must be real values with the first value smaller than the second. Otherwise iscont will return FAIL.

EXAMPLES :

```
readlib(iscont):
iscont( 1/x, x=1..2 );        yields true
iscont( 1/x, x=–1..1 );       yields false
iscont( 1/x, x=0..1 );        yields true
iscont( 1/(x+a), x=0..1);     yields FAIL
```

9.2.27. isolate – isolate a subexpression to left-side of an equation

CALLING SEQUENCE : isolate(eqn,expr); or isolate(eqn,expr,iter);
PARAMETERS : eqn – an equation or an algebraic expression
expr – any algebraic expression
iter – an (optional) positive integer

- The procedure "isolate" attempts to isolate the second argument "expr" in the first argument "eqn" and solves for "expr" in "eqn" .
- If the first argument is not an equation, then the equation eqn=0 is assumed.
- The optional third argument "iter" controls the maximum number of transformation steps that "isolate" will perform.
- This procedure must be read in before it is used, by doing "readlib(isolate):" .

EXAMPLES :

```
isolate(4*x*sin(x)=3,sin(x));   yields  sin(x) = 3/4/x
isolate(x^2–3*x–5,x^2);         yields  x^2 = 3*x+5
```

9.2.28. issqr – test if an integer is a perfect square

CALLING SEQUENCE : issqr(<int>);
PARAMETER : <int> – a positive integer

- The function "issqr" returns true if n is the square of an integer.

- "issqr" must be read in the Maple session initially by typing "readlib(issqr):".

EXAMPLES :

```
issqr(9);    yields  true
issqr(10);   yields  false
```

9.2.29. laplace – Laplace transform

CALLING SEQUENCE : laplace(expr, t, s);

PARAMETERS :
- expr – expression to be transformed
- t – variable "expr" is transformed with respect to
- s – variable to appear in transformed expression

- "laplace" applies the Laplace transformation to "expr" with respect to "t".
- Expressions which are sums of certain special functions or rational functions of polynomials can be transformed.
- When transforming expressions like diff(y(t),t,t), "laplace" will insert the initial values y(0), yp(0), etc. yp(0) is the value of the first derivative at 0, and ypp(0), etc. have the obvious meanings.
- "laplace" and "invlaplace" recognize the Dirac-delta (or unit-impulse) function as Dirac(t).
- This function must be loaded by entering "readlib(laplace):".

EXAMPLES :

```
readlib(laplace):
laplace(t^2+sin(t)=y(t),t,s);    yields
          (s^3+2*s^2+2) / (s^3*(s^2+1)) = laplace(y(t),t,s)
laplace(t^(3/2)-exp(t)+sinh(a*t),t,s);    yields
            3/4*sqrt(Pi)/s^(5/2) - 1/(s-1) + a/(s^2-a^2)
laplace(diff(y(t),t$2)-y(t)=sin(a*t),t,s);    yields
 (laplace(y(t),t,s)*s - y(0))*s - yp(0) - laplace(y(t),t,s) = a/(s^2+a^2)
```

SEE ALSO : invlaplace

9.2.30. latex – produce output suitable for latex printing

CALLING SEQUENCE : latex(expr, filename)

PARAMETERS :
- expr – any expression
- filename – (optional) output to a file

- ``latex'' produces output which is suitable for printing ``expr'' with a latex processor.

- The format of the output depends on the degree of simplification of expr, no attempt is made to modify the input expression. The mathematical format is taken, in general, from the CRC handbook or the Handbook of Mathematical functions.
- The functions sum, int, diff, limit and log are aliased by Sum, Int, Diff, Limit and Log, so that these can be used to prevent evaluation by maple.
- User-defined function printing can be interfaced by including a procedure `latex/<function-name>`.
- ``latex" will not produce the "\[" or "\]" lines in the output file.
- This function has to be loaded via readlib(latex):

EXAMPLES :

```
writeto(LatexFile);
latex( Int( 1/(x^2+1), x) = int( 1/(x^2+1), x) );
writeto(terminal);
latex( x^2 + y^2 = z^2, OtherLatexFile );
```

9.2.31. match – pattern matching

CALLING SEQUENCE : match(expr = pattern, v, 's');

PARAMETERS :

	expr	- the expression to be matched
	pattern	- the pattern to match
	v	- the name of the main variable
	s	- the name of the return argument

- Match returns true if the match was successful, false otherwise. If the match was successful, s is assigned (call by name) a substitution set such that subs(s, pattern) = expr.
- The main variable is a variable which has to be matched exactly into the pattern. In other words, the main variable cannot be substituted for any value.
- This function has to be loaded with the command readlib(match):

EXAMPLES :

match(ln(k)/k^(1/2) = A*ln(k)^P*k^Q, k, 's'); yields true
s; yields {A=1, P=1, Q=-1/2}

match(5*x^2-3*x+z*x+y = a*(x+b)^2+c, x, 's'); yields true
s; yields {a=5, b=-3/10+1/10*z, c=-9/20+3/10*z-1/20*z^2+y}

SEE ALSO : help(solve,identity)

9.2.32. minimize and maximize – compute the minimum (maximum)

CALLING SEQUENCE : minimize(<expr>);
maximize(<expr>);
minimize(<expr>, {vars});
maximize(<expr>, {vars});

PARAMETER : <expr> – any algebraic expression

- The function minimize returns the minimum value of <expr> or an expression sequence with all the values that are candidates for minima.
- When the solution is in terms of algebraic numbers of degree larger than 2, minimize will return a floating point approximation to the minima, as it is not possible to express (at present) a particular branch of a "RootOf(...)" expression.
- The present function finds unconstrained minima over the real numbers, i.e. for any values of {vars} which are real (including the limits at infinity).
- The function maximize is defined to compute the negative of the result of minimize applied to the negated <expr> .
- To use either of these two functions, first type "readlib(minimize):".

EXAMPLES :

```
readlib(minimize):
minimize( x^2 + y^2 + 3 );          yields  3
minimize( sin(x) );                 yields -1
minimize( (x^2-y^2)/(x^2+y^2) );    yields -1
maximize( (x^2-y^2)/(x^2+y^2) );    yields  1
```

SEE ALSO: simplex

9.2.33. mtaylor – multivariate taylor series expansion

CALLING SEQUENCE : mtaylor(t, e1, e2, ..., em, n)

PARAMETERS : t – any algebraic expression
ek – equations (xk = ak) or simply a name xk
n – (optional) non-negative integer

- "mtaylor" computes a multivariate taylor series expansion of "t", with respect to the variables "xk", about the points "ak", up to order "n". If ek is simply a name "xk", the equation "xk = 0" is assumed.
- If the third argument "n" is present then it specifies the "truncation order". The concept of "truncation order" used is that of "total degree" not "vector degree". If "n" is not present, the "truncation

order" is determined by the global variable Order. The user may assign any non-negative integer to Order. The default value of Order is 6.

- mtaylor is a first implementation of multivariate taylor series, it will later be incorporated into the taylor command. Thus far, the facility is fairly primitive. Note, mtaylor restricts its domain to "pure" taylor series, series with non-negative powers in the variables.
- The data structure used is a function with the name "series". The first operand is the order of the series. The second operand is a list of the expansion points, (x1 = a1, x2 = a2, ..., xm = am), and the third operand is a list of the terms. The kth term for k = 0,1,...,n−1 is of total degree k in the variables (x1−a1), (x2−a2), ..., (xm−am).
- This function must first be loaded via: readlib(mtaylor):

EXAMPLES :
```
readlib(mtaylor):
mtaylor( arcsin(x+y), x, y, 4 );
series(4, [x,y], [0, x+y, 0, 1/6*x^3+1/2*x^2*y+1/2*x*y^2+1/6*y^3])
mtaylor( exp(x*y^2+y^2), x, y, 7 );
series(7, [x,y], [1, 0, y^2, x*y^2, y^4/2, x*y^4, x^2*y^4/2+y^6/6])
mtaylor( exp(x)+log(y), x, y=1, 3 );
series(4, [x,y−1], [1, x+y−1, 1/2*x^2−1/2*(y−1)^2, 1/6*x^3+1/3*(y−1)^3])
```
SEE ALSO : series, taylor

9.2.34. optimize – common subexpression optimization

CALLING SEQUENCE : optimize(e, 'f', p);
PARAMETER : e – an expression or an array
f – (optional) name
p – (optional) formal parameters

- This function must first be loaded via: readlib(optimize):
- The optimize function outputs a sequence of equations of the form t.k = expr which represent an optimized evaluation sequence for input expression "e".
- The second argument "f" is the name to which a Maple procedure which computes this optimized evaluation sequence will be assigned (call–by–name). Note that a temporary file named "optemp" is created. The third argument "p" is used to specify which of the names in "e" are to be formal parameters in "f". "p" may be a single name or a list / set of names.

- Optimize makes use of Maple's option remember facility to identify common subexpressions in linear time/space. This means, however, that only those subexpressions for which maple has simplified to be identical are found. For example, the expression x+y is not recognized as being common to x+y+z. That is, optimize performs only "syntactic" optimizations.

EXAMPLES :

```
readlib(optimize):
optimize(x^4);              yields  t1 = x^2, t2 = t1^2
optimize(2*x^3-y*x^4);      yields  t1 = x^2, t3 = t1^2, t5 = 2*x*t1-y*t3
```

9.2.35. permute – compute all permutations of a sequence

CALLING SEQUENCE : permute(N); permute(N, r);

PARAMETERS : N – a list of elements or an integer
r – integer (by default r = nops(N))

- If N is a list, then "permute" returns a list of all the permutations of the elements taken r (default r := nops(N)) at a time.
- An integer parameter for N is equivalent to the list of integers from 1 to N.
- "permute" must be read in before using it, by typing: "readlib(permute):" .

EXAMPLES :

```
permute([a,b,c]);
    [[a,b,c], [a,c,b], [b,a,c], [b,c,a], [c,a,b], [c,b,a]]
permute([a,b,c], 2);
    [[a,b], [a,c], [b,a], [b,c], [c,a], [c,b]]
permute(3);
    [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
permute(3, 2);
    [[1,2], [1,3], [2,1], [2,3], [3,1], [3,2]]
```

9.2.36. priqueue – Priority Queue package

CALLING SEQUENCE : initialize(pq);
insert(NewRecord, pq);
extract(pq);

PARAMETERS : pq – the name of the priority queue
(multiple priority queues are allowed)
NewRecord – the structure to be inserted in the queue

- ``initialize" makes the necessary assignments (side effects) so that the queue ``pq" is initialized with an empty queue. ``pq" is treated as an unbounded array of the inserted records. Additionally, pq[0] maintains the number of entries in the queue.
- ``insert" adds an element to the queue. The queue is a maximum queue, that is, the maximum element is kept on the head of the queue. The criteria for ordering the queue is giving by ``NewRecord[1]" and hence this value should exist and be comparable.
- ``extract" returns the stored record with the highest value (in its first position) and deletes this record from the queue.
- If inspection alone is desired, the maximum element of the queue is always at ``pq[1]".
- These functions have to be loaded with the command: readlib(priqueue):

EXAMPLES :
initialize(DailyTemps);
insert([14,Apr30], DailyTemps);
insert([24,May4], DailyTemps);
extract(DailyTemps); yields [24,May4]
extract(DailyTemps); yields [14,Apr30]

9.2.37. profile, unprofile, showprofile – space & time profile of a proc

CALLING SEQUENCE : profile(f1,f2, ...);
unprofile(f1,f2, ...);
showprofile();
PARAMETERS : f1,f2,... – the names of the procedures to be profiled

- These functions must first be loaded via: readlib(profile):
- The functions "profile", "unprofile" and "showprofile" are for profiling Maple procedures. To find out how much space and time some procedures are using, issue the command: "profile(f1,f2,...);" then perform whatever calculations you wish to have profiled. To view the results, use: "showprofile();" . To return the procedures to their normal unprofiled state, do: "unprofile(f1,f2,...);".
- Both profile and unprofile return the names of the procedures that they were invoked with. "showprofile" returns NULL and prints the time and space results.

SEE ALSO : time, showtime, trace

9.2.38. psqrt and proot – square root and nth root of a polynomial

CALLING SEQUENCES : psqrt(p); and proot(p,n);
PARAMETER : p – multivariate rational coefficient polynomial
n – integer

- If p is a perfect square, psqrt returns a square root of p. Otherwise it returns the name _NOSQRT.
- If p is an nth power, proot(p,n) returns the nth root of p. Otherwise proot returns the name _NOROOT.
- Both functions must first be loaded in by readlib.

EXAMPLES :

```
readlib(psqrt):
readlib(proot):
psqrt( 9 );                          yields  3
proot( 9, 2 );                       yields  3
psqrt( x^2 + 2*x*y +y^2 );           yields  x+y
proot( x^3 + 3*x^2 + 3*x + 1, 3 );   yields  x + 1
psqrt( x+y );                        yields  _NOSQRT
proot( x+y, 2 );                     yields  _NOROOT
```

SEE ALSO : isqrt, iroot

9.2.39. randpoly – random polynomial generator

CALLING SEQUENCE : randpoly(vars, eqns);
PARAMETERS : vars – indeterminate or list or set of indeterminates
eqns – (optional) equations specifying properties

- This function must first be loaded via: readlib(randpoly):
- Randpoly generates a random polynomial in vars. It is useful for generating test problems for debugging, testing and demonstration purposes. Several options can be specified, determining the form of the polynomial. This allows for quite general expressions with certain properties.
- The first argument vars specifies the variables in which the polynomial is to be generated. It is typically a single variable, (or more generally any algebraic expression) or, a list or set of such (for generating multivariate polynomials).
- Remaining arguments are interpreted as options. They are input as equations in any order. The lhs identifies the particular option. The rhs is an expression or procedure which is to be evaluated. The rhs could be an unevaluated function call to randpoly (for generating

recursive structures) or a random number generator.

The options (and their default values) are:

coeffs	generate the coefficients	rand(–99..99)
expons	generate the exponents	rand(6)
terms	number of terms generated	6
degree	total degree for a dense polynomial	5
dense	the polynomial is to be dense	

EXAMPLES :

```
readlib(randpoly):
randpoly( x );
randpoly( [x,y], terms=20 );
randpoly( [x,sin(x),cos(x)] );
randpoly( z, expons=rand(–5..5 ));
randpoly( [x], coeffs='randpoly([y])' );
randpoly( [x,y], dense );
```

9.2.40. realroot – isolating intervals for real roots of a polynomial

CALLING SEQUENCE : realroot(poly,widthgoal)

PARAMETERS : poly – a univariate polynomial with integer coeffs
widthgoal – (optional) max. size of each isolating interval

- This function must be loaded via readlib(realroot):
- realroot(poly, widthgoal) returns a list of isolating intervals for all real roots of the univariate polynomial poly. The width of the interval is less than or equal to the optional parameter "widthgoal", a positive number. If widthgoal is omitted, the most convenient width is used for each interval returned.
- The intervals are specified as a list of two rational numbers. The list/interval [x,x] means the single point "x". [a,b] means the open interval (a,b). Multiplicity information is not included.
- realroot uses Descartes' rule of signs (see Loos and Collins in "Computer Algebra", B. Buchberger, ed.).

EXAMPLES :

realroot(x^8+5*x^7–4*x^6–20*x^5+4*x^4+20*x^3,1/1000); yields
[[0, 0], [181/128, 1449/1024], [–1449/1024, –181/128], [–5, –5]]

9.2.41. replot – redo a plot

CALLING SEQUENCE : replot(p, h, v, ...);
PARAMETERS : p – a plot data structure
h – horizontal range (optional)
v – vertical range (optional)
... – optional arguments (see below)

- This function allows one to change the parameters of a plot without the plot being recalculated. This allows one to zoom in on a plot or change the title, labels, or style. The options to replot are the same as the options to plot. See help(plot,`options`);
- Note that since replot doesn't recompute the function, it won't provide more data in a new plot than was in the original plot. So changing the scale won't show more of the function or increase any detail. This requires computing the plot again using plot. Also, replot can't switch between infinity and bounded plots.
- Note: If a vertical range is specified, the horizontal range must be specified too, since the first range is assumed to be the horizontal range.
- This function must first be loaded via: readlib(replot):

EXAMPLES :

```
readlib(replot):
plot(tan(x),0..Pi);                  yields a plot of tan(x)
replot(*,x=0..Pi,y=-5..5);           yields a clipped plot of tan(x)
replot(*,x=0..Pi/2,title=tan);       yields a plot of tan(x) on 0..Pi/2
p := plot(sin(x),0..2*Pi):           p contains a plot of sin(x)
q := replot(p,x=0..Pi,y=-1..1):      q contains a plot with labelled axes
q;                                   yields the plot of sin(x)
```

SEE ALSO : plot

9.2.42. residue – compute the algebraic residue of an expression

CALLING SEQUENCE : residue(f, x=a)
PARAMETERS : f – an arbitrary algebraic expression
x – a variable in ``f''
a – an algebraic value at which the residue is evaluated

- Computes the algebraic residue of the expression ``f'' for the variable ``x'' around the point ``a''. The residue is defined as 1/(2*Pi*I) * int( f/(x–a) ) where the integration path is a circle around ``a'' in the complex plane. An alternative definition of the residue is the

coefficient of (x–a)^(–1) in the Laurent expansion of ``f''.

- This function has to be loaded with the command: readlib(residue):

EXAMPLES :
residue(Zeta(s), s=1); yields 1
residue(Psi(x)*GAMMA(x)/x, x=0); yields gamma^2/2 + Pi^2/12
residue(exp(x), x=1); yields 0
SEE ALSO : taylor, singular

9.2.43. S – the Fresnel sine integral

The Fresnel sine integral is defined by

$$S(x) = \int_0^x \sin(\frac{\pi}{2}t^2)\ dt\ .$$

If this function is read in via "readlib(S):" then simplifications will be applied.

9.2.44. search – substring search

CALLING SEQUENCE : search(s,t); or search(s,t,'p');
PARAMETERS : s,t – strings
p – name

- The search function is a boolean valued function that searches the subject string "s" for the first occurrence of the target string "t".
- The third argument if present will be assigned the index where the target string starts, if the search was successful.
- This function must first be loaded via "readlib(search)".

EXAMPLES :

```
search(abcxxefg,xx,'p');   yields  true
p;                         yields  4
```

SEE ALSO : substring

9.2.45. select – selection from a list, set, sum, or product

CALLING SEQUENCE : select(f,expr); or select(f,expr,b1,...,bN);
PARAMETERS : f – a boolean-valued procedure
expr – a list, set, sum, or product
b1,...,bN – optional extra arguments

- The function "select" keeps the operands of expr which satisfy the boolean-valued procedure f, creating a new object of the same type as expr. Additional arguments b1,...,bN are passed to f.
- This function must first be read in by typing "readlib(select):" .

EXAMPLES :

```
readlib(select):
select( isprime, [$10..20] );          ==> [11, 13, 17, 19]
select( has, 2*x*exp(x)*ln(y), x );    ==> x*exp(x)
f := indets( 2*exp(a*x)*sin(x) );      ==> f := {x, a, exp(a*x), sin(x)}
select( type, f, name );               ==> {x, a}
```

9.2.46. showtime – display time / space statistics for commands

CALLING SEQUENCE : showtime();

- This function is first read in by typing "readlib(showtime):"
- "showtime" is used to maintain a history of results computed within a Maple session and to display the time and space used by Maple for the execution of each statement.
- After loading in "showtime" which defines the two global variables "on" and "off", typing "on;" initiates the history mechanism. The user is prompted initially with the prompt

  I1 ?

  and for successive statements by the prompts numbered I2, I3, ... Any Maple statement entered is evaluated normally, its result returned followed by a line numbered O1, O2, ... with the time taken and the amount of memory used being displayed. The result is assigned to the global variable O1, O2, ... which may be referred to later by the user thus providing a complete history mechanism.
- By typing "off;" Maple returns to normal mode using the standard prompt.

SEE ALSO : time, profile

9.2.47. Si – the sine integral

The sine integral is defined by

$$\mathrm{Si}(x) = \int_0^x \frac{\sin(t)}{t}\,dt \ .$$

If this function is read in via "readlib(Si):" then simplifications will be applied.

9.2.48. singular – find all singularities of an expression

CALLING SEQUENCE : singular(<expr>);
singular(<expr>, <var>);
singular(<expr>, <set–of–vars>);

PARAMETERS :
<expr> – an algebraic expression
<var> – a variable
<set–of–expr> – a set of variables

- Singular outputs an expression sequence of points where singularities occur. Each point is defined by a set of equations, the left hand sides of the equations being the independent variables.
- Singularities are found for values of the independent variables. The independent variables are the variable or set of variables given or the indets(expr,name) if not given. All the other variables are considered constants.
- Singular will return non-removable as well as removable singularities. I.e. (x–2)/sin(x–2) will report a singularity at x=2.
- The power of singular to find singularities is basically that of solve. For example, some zeros that solve cannot find may result in singularities that singular will not find.
- Singular may return expressions in _N or in _NN. _N is assumed to be variable taking arbitrary integer values. _NN is assumed to be an arbitrary natural number.
- This function has to be loaded with the command: readlib(singular);

EXAMPLES :

```
singular( x*y + 1/(x*y), x );    {x = 0}, {x = infinity}
singular( ln(x)/(x^2–1) );       {x = 0}, {x = 1}, {x = –1}
singular( x/(x–y) );             {y = y, x = y}
singular( tan(x) );              {x = _N*Pi+1/2*Pi}
singular( Psi(1/x) );            {x = 0}, {x = infinity}, {x = –1/(_NN–1)}
```

9.2.49. sinterp – sparse multivariate modular polynomial interpolation

CALLING SEQUENCE : sinterp (f, x, t, p)

PARAMETER :
f – a Maple procedure
x – a list of variables to be interpolated
t – a positive integer
p – prime modulus

- Given a procedure "f" that evaluates a polynomial at values for the variables in the list "x" modulo "p", sinterp interpolates the polynomial modulo "p" using sparse interpolation .
- For the interpolation to succeed, the modulus "p" should be sufficiently large. If it is not, FAIL is returned.
- The degree bound "t" is a bound on the total degree of the polynomial being interpolated.
- This function must be loaded via: readlib(sinterp):

SEE ALSO : sdegree, interp

9.2.50. stir1 and stir2 – Stirling numbers of the first and second kind

CALLING SEQUENCES : stir1(n,m); and stir2(n,m);
PARAMETERS : n, m – any integers

- "stir1" computes the Stirling numbers of the first kind S(n,m). (–1)^(n–m) S(n,m) is the number of permutations of n symbols which have m cycles.
- "stir2" computes the Stirling numbers of the second kind. stir2(n,m) is the number of ways of partitioning a set of n elements into m non-empty subsets.
- These function must be loaded via "readlib(stir1):" and "readlib(stir2):".

9.2.51. symmpoly – determine whether a(x) is a symmetric polynomial

CALLING SEQUENCE : symmpoly(a, x, 'p');
PARAMETERS : a, x polynomial in x
p optional

- Determine whether a(x) is a symmetric polynomial. a is symmetric if and only if coeff(a,x,k) = coeff(a,x,d–k) for all k = 0..d where d = degree(a,x).
- If d is even and if the optional second argument p is specified, p is assigned the polynomial of degree d/2 such that x^(d/2) p(x+1/x) = a .
- Note that if d is odd and a is symmetric, a is divisible by x+1 . In this case, if p is specified then the result computed is for a / (x+1).
- This function must be loaded via readlib(symmpoly):

EXAMPLES :
symmpoly(1+2*x+2*x^3+x^4,x,p); yields true
p; yields x^2+2*x-2

9.2.52. totorder – package for total ordering of names

CALLING SEQUENCES : assume(r), is(r), forget(r), ordering()
PARAMETERS : r – sequence of relations

- This package of functions must be defined in a Maple session by typing: "with(totorder):".
- "assume" allows a total ordering to be defined on a sequence of names.
- Having defined such an ordering, the "is" function can be used to make queries with respect to the ordering.
- The "forget" function deletes relationships within the ordering. To delete all relationships do: forget(everything); .
- The "ordering" function prints out the current total ordering.

EXAMPLES :

```
with(totorder):
assume( a<b, b<c, c=d, d<f );
assume( u>b );
is( u<f );                      yields  true
forget(u);
ordering();                     a<b, b<c, c=d, d<f
assume(u>d);
is( c/d-1=0 );                  yields  true
is( 2*c-c<f );                  yields  true
is( c<g );                      yields  c<g
```

9.2.53. translate – linear translation of a polynomial

CALLING SEQUENCE : translate(a, x, x0);
PARAMETERS : a, x – a is polynomial in x
x0 – a constant

- Translate a(x) by x = x + x0 efficiently. The method used requires O(n) multiplications and divisions and O(n^2) additions where n = degree(a,x).
- This function must be loaded via: readlib(translate):

9.2.54. trigsubs – handling trigonometric identities

CALLING SEQUENCE : trigsubs(e); trigsubs(s, <expr>);
PARAMETERS : e – any expression or equation
s – any equation
<expr> – any expression

- First, this procedure must be read in with: "readlib(trigsubs):"
- "trigsubs" returns a set of functions known to the procedure (one arg 0), or a list of trig identities known to the procedure for the argument (one expression arg), or a string which indicates if the identity was found (one equation arg), or an expression with 'found' trig identity substituted (two args).
- To create trig identities, use "trigsubs(e)" to obtain the list of equivalent expressions known to the procedure then use "op" to construct the desired identity.
- For substitution of identities not known to this procedure use "subs".

EXAMPLES :

```
trigsubs(0);                          yields  {sin,cos,tan,sec,csc,cot}
trigsubs( cos(a+b*w) );               yields  [cos(a+b*w),cos(-a-b*w),...]
trigsubs( cos(w) = sin(w) );          yields  `not found`
trigsubs( cos(w) = sin(w), 1 );       yields  Error ..., not found ...
trigsubs( sin(2*z) = 2*cos(z)*sin(z), sin(2*z)*cos(z) );
                                      yields  2*cos(z)^2*sin(z)
```

9.2.55. unapply – returns an operator from an expression and arguments

CALLING SEQUENCE : unapply(expr,x,y,..)
PARAMETERS : expr – any expression
x,y,.. – variable names

- The result of unapply(expr,x) is a functional operator.
- unapply(expr,x)(x) ==> expr .
- In particular, for a function f(x), unapply(f(x),x) ==> f .
- Whenever it is desired to construct an operator using contents of variables or evaluated expressions, unapply should be used.
- This function implements the lambda-expressions of lambda calculus.
- This function must first be loaded using "readlib(unapply):"

EXAMPLES :

```
readlib(unapply):
p:= any polynomial
```

unapply(p,x); yields p as an operator
unapply(p,x)(x); yields the original p
SEE ALSO :"An Implementation of Operators for Symbolic Algebra Systems" by G.H. Gonnet, SYMSAC July 1986, help(operators,<topic>), for <topic> functional, example or D.

9.2.56. write, writeln, open, close

CALLING SEQUENCE : write(r), writeln(r), open(f), close()
PARAMETERS : r – sequence of expressions, possibly empty
f – name of a file

- These functions are read into a Maple session by typing: "readlib(write):".
- The user would typically open a particular file for output by open(filename), write some expressions to that file using write and writeln, then close the file by close(). Expressions are line printed to the file. A buffer is maintained so that only after the output exceeds the line length (screenwidth) or a writeln (for write followed by new line) or a close, is the buffer flushed.
- Caveats: Maple does not allow more than one file to be open for input or output at one time.

SEE ALSO : writeto

9.2.57. zip – zip up two or more lists or vectors

CALLING SEQUENCE : zip(f,u,v,...)
PARAMETERS : f – binary function
u,v,... – lists or vectors

- The function "zip" is used to apply the binary function "f" to two lists/vectors "u" and "v" creating a new list/vector "r" where r[i] = f(u[i],v[i]) for i up to the length of the shorter of "u" and "v".
- If there is only one list/vector, it is returned. If there are more than two lists/vectors, the rule zip(f,u,v,w) = zip(f,zip(f,u,v),w) is applied.
- This function must first be read in by typing "readlib(zip):" .

EXAMPLES :
zip(<x+y>,[1,2,3],[4,5,6]); yields [5,7,9]
zip(gcd,[0,1,2],[2,4,6]); yields [2,1,2]

9.3. The `difforms` package for Differential Forms

SYNTAX : difforms[<function>] (args); or <function> (args);

- To use a difforms function, type "with(difforms, <function>);" to define <function>, or type "with(difforms);" to define all difforms functions.
- The functions available are:

```
&^          d          defform     formpart     mixpar
parity      scalarpart simpform    wdegree
```

- For example, to take the exterior derivative of an expression:

```
with(difforms);
 [mixpar, d, formpart, wdegree, scalarpart, simpform, parity, &^, defform]
defform(w1=1,w2=1,w3=1,f=scalar,g=scalar,C=const);
d( f*w1^2+g*&^(w2,w1)+f*&^(w2,w3) );
      &^(d(f), w1^2) + &^(d(g), w2, w1) + g &^(d(w2), w1)
        - g &^( w2, d(w1)) + &^(d(f), w2, w3)
        + f &^(d(w2), w3) - f &^(w2, d(w3))
```

- For help with a particular function do: help(difforms, <function>);
- For help with the basic types used by the package (const, scalar, form), do: help(difforms, const); or help(difforms, scalar); or help(difforms, form);

SEE ALSO : with

9.3.1. type/const – extension of type/constant

CALLING SEQUENCE : type(<expr>, const);
PARAMETERS : <expr> – maple expression

- type/const includes any <expr> which is type/constant, plus those names which are defined to be const through defform.
- All sums, products, or powers of objects of type/const are considered to be type/const.
- type/const, type/form and type/scalar are mutually exclusive. Any expression should belong to at most one of these types.

EXAMPLES :

```
with(difforms): defform(a=const,b=const,c=const,d=const);
type(a,const);       yields    true
type(a+b*c^d,const); yields    true
type(e+a,const);     yields    false
```

9.3.2. d – exterior differentiation

CALLING SEQUENCE : d(<expr>) or d(<expr>, <forms>)
PARAMETERS : <expr> expression or list of expressions
<forms> (optional) list of 1-forms, i.e. wdegree=one

- "d" computes the exterior derivative of an expression. If the expression is a list, then "d" is applied to each element in the list.
- When d is called with an expression and a list of 1-forms, any name of type scalar in the expression will be expanded in these 1-forms. It is assumed that the 1-forms are independent. For each 1-form, a new scalar is created to be the component for that 1-form.

EXAMPLES :
```
with(difforms): defform(f=0,w1=1,w2=1,w3=1,x=0,y=0,z=0);
d( x^2*y );        yields  2*x*y*d(x) + x^2*d(y)
d( f(x,y,z) );     yields
   diff(f(x,y,z),x)*d(x)+diff(f(x,y,z),y)*d(y)+diff(f(x,y,z),z)*d(z)
d( f, [w1,w2,w3] ); yields   fw1*w1+fw2*w2+fw3*w3
d( f*&^(w1,w2) );  yields
        &^(d(f),w1,w2) + f * &^(d(w1),w2) – f * &^(w1,d(w2))
d( [x*y,y^2,w2] ); yields
             [ y*d(x)+x*d(y), 2*y*d(y), d(w2) ]
```
SEE ALSO : help(difforms,mixpar)

9.3.3. defform – define a constant, scalar or form

CALLING SEQUENCE : defform(<n1> = <e1>, <n2> = <e2>, . . .)
PARAMETERS : <n1>, <n2>, ... – names or calls to d
<e1>, <e2>, ... – maple expressions

- "defform" is used to define the basic variables used in a computation, or to define the exterior derivative of an expression.
- "defform" clears the remember tables of all functions in the forms package, as changing the definition of a form can make the remembered results invalid. However, definitions made through defform are not cleared; they are remembered permanently.
- "defform" takes an arbitrary number of equations, where each equation is <name> = <expression>. There are certain expressions – const, scalar, form, odd, even, –1, and 0 – that have special meanings. Except for these, <name> = <expression> means <name> is a form and wdegree(<name>) = <expression>.

- "const" or "–1" means type(<name>,const) is true. "even" or "odd" means type(<name>,const) is true, but also that parity(<name>) is 0 or 1, as appropriate. "even" and "odd" are useful for specifying a form with even or odd, but otherwise unknown wdegree.
- "scalar" or "0" means type(<name>,scalar) is true. "form" means that type(<name>,form) is true, but does not give a value to wdegree(<name>).
- "defform" can be used to define the exterior derivative of an expression, and these derivatives are remembered permanently.

EXAMPLES :
```
with(difforms):
defform(a=const,b=scalar,e=nonhmg,j=3,f=odd,l=f^3,d(l)=e);
type(a,const);          yields    true
type(b,scalar);         yields    true
type(e,form),wdegree(e); yields    true, nonhmg
type(j,form),wdegree(j); yields    true, 3
type(f,const),parity(f); yields    true, 1
d(l);                   yields    e
```

9.3.4. type/form – check for forms

CALLING SEQUENCE : type(<expr>, form);
type(<expr>, form, <int>);
PARAMETERS : <expr> – maple expression
<int> – (optional) positive integer

- type(<expr>,form,<int>) is true if type(<expr>,form) is true and wdegree(<expr>)=<int>, i.e. <expr> is an <int>-form
- type/form includes all expressions declared to be forms through defform, as well as any product or sum that contains at least one term/factor of type/form, and a power is of type/form if the base is of type/form
- type/const, type/scalar, and type/form are mutually exclusive. Any expression can belong to at most one of these types.

EXAMPLES :
```
with(difforms): defform(a=const,b=scalar,f=form,g=a);
type(a+b+f, form);     yields    true
type(a+b, form);       yields    false
type(f^a, form);       yields    true
type(a*g, form);       yields    true
```

9.3.5. formpart – find part of an expression which is a form

CALLING SEQUENCE : formpart(<expr>);
PARAMETERS : <expr> – any maple expression, but usually a product

- "formpart" is most effective on products. It will try to simplify other expressions to products.
- This function removes scalars and constants from a product, leaving only that part which is a form, or is undeclared.
- This function and scalarpart can be used to separate a product into form and scalar parts.

EXAMPLES :
```
with(difforms): defform(f=scalar);
formpart( 2*&^(f*v,u) );  yields  &^(u,v)
```

9.3.6. mixpar – ensure equality of mixed partial derivatives

CALLING SEQUENCE : mixpar(expr);
PARAMETERS : expr – maple expression

- "mixpar" will take nested calls to "diff", and sort the sequence of differentiations so that they are sorted by lexicographical ordering on the variables that are being differentiated with respect to.
- The purpose of the this function is ensure that equal mixed partials are recognized as equal.

EXAMPLES :
```
with(difforms): defform(f=0,x=0,y=0,z=0);
d(f(x,y,z));              yields
 diff(f(x,y,z),z)*d(z) + diff(f(x,y,z),y)*d(y) + diff(f(x,y,z),x)*d(x)
d( " );                   yields
(diff(diff(f(x,y,z),z),y)-diff(diff(f(x,y,z),y),z))*&^(d(y),d(z))+
(diff(diff(f(x,y,z),z),x)-diff(diff(f(x,y,z),x),z))*&^(d(x),d(z))+
(diff(diff(f(x,y,z),y),x)-diff(diff(f(x,y,z),x),y))*&^(d(x),d(y))
mixpar( " )               yields
                              0
```
SEE ALSO : diff, sort

9.3.7. parity – extension of mod 2

CALLING SEQUENCE : parity(<expr>);
PARAMETERS : <expr> – maple expression

- "parity" extends mod 2 for the difforms package, by assuming that unspecified exponents are integers, which is true in the package but not in general. Given n an integer, one can use p^n mod 2 = p.
- It is also through the parity function that defform specifies certain names as even or odd.

EXAMPLES :
```
with(difforms): defform(w=p,p=even);
parity(3*p);          yields     0
parity(p^m);          yields     0
```
SEE ALSO : `mod`

9.3.8. type/scalar – check for scalars

CALLING SEQUENCE : type(<expr>, scalar);
PARAMETERS : <expr> – maple expression

- type/scalar includes all names defined to be type/scalar through defform.
- Any sum or product which contains one term/factor that is type/scalar, and the remaining terms/factors are all either type/scalar or type/const, is considered to be type/scalar.
- Any power, where the base is type/scalar, is considered to be type/scalar.
- type/const, type/scalar, and type/form are mutually exclusive. Any expression can belong to at most one of these types.

EXAMPLES :
```
with(difforms): defform(a=const,b=const,c=scalar,d=scalar);
type(c+d^3,scalar);  yields    true
type(a+d,scalar);    yields    true
type(a+d+f,scalar);  yields    false
type(a+b,scalar);    yields    false
```

9.3.9. scalarpart – find part of an expression which is a scalar

CALLING SEQUENCE : scalarpart(<expr>);
PARAMETERS : <expr> – maple expression, but usually a product

- "scalarpart" is most effective on products. It will try to simplify other expressions to products.
- This function collects scalars and constants from a product, removing those parts which are forms, or are undeclared.

- This function and formpart may be used together to separate a product into scalar and form parts.

EXAMPLES :
```
with(difforms): defform(f=scalar,c=const);
scalarpart( 2*&^(v*f,c*u) );  yields  2*f*c
```

9.3.10. simpform – simplify an expression involving forms

CALLING SEQUENCE : simpform(<expr>);
PARAMETERS : <expr> – maple expression

- "simpform" will simplify an expression involving forms. It will collect like terms, simplify wedge products, pull out scalar factors, etc.

EXAMPLES :
```
with(difforms): defform(f=scalar, g=scalar);
&^(f*v,u)+&^(u,g*v); yields f * &^(v,u) + g * &^(u,v)
simpform( " );       yields (f+(-1)^(wdegree(u)*wdegree(v))*g)*&^(v,u)
f*(&^(u,v)+&^(u,w))+g*&^(u,v):
simpform( " );       yields (f + g) * &^(u,v) + f * &^(u,w)
```

9.3.11. wdegree – degree of a form

CALLING SEQUENCE : wdegree(<expr>);
PARAMETERS : <expr> – maple expression

- "wdegree" computes the degree of an expression, considered as a form.
- "wdegree" returns nonhmg when the expression is a sum of forms which have different degrees.
- Expressions of type scalar or const have wdegree = 0.

EXAMPLES :
```
with(difforms): defform(x=p, c=const);
wdegree( c );          yields  0
wdegree( x );          yields  p
d(x^2);               yields  (1 + (-1)^p) * &^(d(x),x)
wdegree( " );          yields  2*p + 1
wdegree( x + d(x) );   yields  nonhmg
```

9.3.12. &^ – wedge product

CALLING SEQUENCE : &^(<expr1>, <expr2>, ...); or
<expr1> &^ <expr2> &^ ...

PARAMETERS : <expr1>, <expr2>, ... – maple expressions

- "&^" represents the wedge product of differential forms.
- Elementary simplifications are done on wedge products, for example, if "a" is a form of odd degree, then &^(a,a) is simplified to 0.
- "&^" will distribute over "+" whenever possible. The preferred representation of "&^" is a sum of wedge products. Otherwise, it may be necessary to apply "expand", then "simpform" to an expression to reduce it to simplest form.

EXAMPLES :

```
with(difforms): defform(a=1,b=1,c=1,d=2,e=2);
&^(a^2,d,e)                yields    &^(a^2, d, e)
&^(a,b,c+&^(d,e));         yields    &^(a, b, c) + &^(a, b, d, e)
&^(a,b,c+&^(d,e,&^(a,d))); yields    &^(a, b, c)
```

9.4. The `grobner` package for Grobner Bases

SYNTAX: grobner[<function>] (args); , or <function> (args);

- To use a grobner function, type "with(grobner, <function>);" to define <function>, or type "with(grobner);" to define all grobner functions.
- The functions available are:

  gbasis normalf leadmon spoly
- The available term orderings are:

  plex – pure lexicographic, where X = [x1, x2, ..., xn]
  induces x1 > x2 > ... > xn ;
  tdeg – total degree (graduated), with ties broken
  by inverse lexicographical order.
- For help with a particular function do: help(grobner, <function>);
- For example, to find the Grobner basis of a set of polynomials "F" over the variables X = [x1,x2 ,..., xn], with respect to the total degree ordering, type:

  with(grobner,gbasis); gbasis(F, X, tdeg);

  If the name `gbasis` were already in use in the current session, one would instead type:

  grobner[gbasis](F, X, tdeg);

SEE ALSO : with

9.4.1. gbasis – compute reduced, minimal Grobner basis

CALLING SEQUENCE: gbasis(F,X); or gbasis(F,X,termorder); or gbasis(F,X,termorder,'Y');

PARAMETERS:

F	– set or list of polynomials
X	– set or list of indeterminates (not including parameters)
termorder	– term ordering: `plex` (pure lexicographic) or `tdeg` (total degree – default)
Y	– (a name) list of reordered indeterminates (output only – required if X is a set)

- gbasis(F,X,termorder) computes the reduced, minimal Grobner basis of the polynomials F with respect to the indeterminates X and the given term ordering. If X is a set, reordering will occur and a fourth argument MUST appear.

- A list of indeterminates X := [x1, x2 ,..., xn] induces the ordering x1 > x2 > ... > xn .
- If the indeterminates are given as a set X := {x1, x2 ,..., xn}, they are automatically reordered to be "heuristically optimal" (in the sense of Boge, Gebauer, Kredel), and the new ordering is assigned to the fourth parameter; a list of indeterminates will not be reordered.

EXAMPLES :
```
with(grobner):
F := [x^2 - 2*x*z + 5, x*y^2 + y*z^3, 3*y^2 - 8*z^3]:
gbasis(F,[y,x,z],plex);          yields
[3*y^2-8*z^3, 80*y*z^3-3*z^8+32*z^7-40*z^5, x^2-2*x*z+5,
 -96*z^7+9*z^8+120*z^5+640*x*z^3, -240*z^6-1600*z^3+96*z^8-9*z^9]
gbasis(F,{x,y,z},plex,'Y');      yields
[x^2-2*x*z+5, 96*z^7-9*z^8-120*z^5-640*x*z^3, 3*y^2-8*z^3,
 40*z^5+3*z^8-32*z^7-80*y*z^3, -240*z^6-1600*z^3+96*z^8-9*z^9]
Y;                               yields   [x, y, z]
gbasis(F,[x,y,z],tdeg);          yields
[x^2-2*x*z+5, -3*y^2+8*z^3, 8*x*y^2+3*y^3, 48*z*y^3+9*y^4+320*y^2]
```
SEE ALSO : help(grobner,normalf)

9.4.2. leadmon – compute the leading monomial of a polynomial

CALLING SEQUENCE: leadmon(poly,vars,termorder); or leadmon(poly,vars);

PARAMETERS:

X	– a list of indeterminates (variable names)
poly	– a polynomial over X
termorder	– (optional) term ordering: either `plex` (pure lexicographic), or `tdeg` (total degree – default)

- leadmon(poly,vars,termorder) computes the leading monomial of "poly" with respect to the chosen term ordering and given indeterminates. It returns a list of two elements; the first is the leading coefficient, and the second is the leading term of "poly".

EXAMPLES :
```
with(grobner):
p := 320*x*y^2 + 9*x*y^4 - 96*z^2*y^4*x + 1600*y^3 - 18*y^5*x*z
       - 592*x*z*y^3 + 45*y^5 + 240*z*y^4:
h := leadmon(p,[x,y],plex);      yields
  h := [-18*z,x*y^5]
h := leadmon(p,[x,y,z],plex);    yields
  h := [-18,x*y^5*z]
h := leadmon(p,[z,y,x]);         yields
  h := [-96,z^2*y^4*x]
```
SEE ALSO : help(grobner)

9.4.3. normalf – reduced form of a polynomial modulo an ideal

CALLING SEQUENCE: normalf(poly,F,X); or normalf(poly,F,X,termorder);

PARAMETERS:

poly	– the polynomial to be reduced
F	– list of polynomials (normally, a Grobner basis)
X	– list of indeterminates (not including free parameters)
termorder	– (optional) term ordering: either `plex` (pure lexicographic), or `tdeg` (total degree – default)

- normalf(poly,F,X,termorder) computes the fully reduced form of "poly" with respect to the ideal basis F, indeterminates X, and term ordering termorder (either `plex` for pure lexicographic ordering, or `tdeg` for total degree ordering).
- Usually, one first computes a Grobner basis for a set of polynomials (algebraic equations, or side relations) via grobner[gbasis], for use as the reducing basis F. Note: F must be a Grobner basis with respect to X,termorder (and not with respect to some permutation of X, for example).
- normalf yields a canonical form for polynomials modulo the ideal generated by F if F is a Grobner basis.

EXAMPLES :
```
with(grobner):
F := [x^2 - 2*x*z + 5, x*y^2 + y*z^3, 3*y^2 - 8*z^3]:  X := [x,y,z]:
G := gbasis(F,X);    yields
 G := [x^2-2*x*z+5, -3*y^2+8*z^3, 8*x*y^2+3*y^3,
                    48*z*y^3+9*y^4+320*y^2]
p := 320*x*y^2 + 9*x*y^4 - 96*z^2*y^4*x + 1600*y^3 - 18*y^5*x*z
      - 592*x*z*y^3 + 45*y^5 + 240*z*y^4:
normalf(p,G,X);    yields  0
q := 3*x^3*y*z^2 - x*z^2 + y^3 + y*z:
normalf(q,G,X);    yields
 - 4*x*z^2 - 41*y^3 + 4*z*y - 60*y*z^2*x + 36*z^2*y^3 + 240*y^2*z
```
SEE ALSO : help(grobner,gbasis)

9.4.4. spoly – compute the S-polynomial of two polynomials

CALLING SEQUENCE: spoly(p1,p2,vars,termorder); or spoly(p1,p2,vars);

PARAMETERS:

	X	– a list of indeterminates (variable names)
	p1,p2	– polynomials over X
	termorder	– (optional) term ordering: either `plex` (pure lexicographic), or `tdeg` (total degree – default)

- spoly(p1,p2,vars,termorder) computes the S–polynomial of "p1", "p2" with respect to the chosen term ordering and given indeterminates.

EXAMPLES :
```
with(grobner):
p := x - 13*y^2 - 12*z^3:
q := x^2 - x*y + 92*z:
spoly(p,q,[x,y,z],plex);    yields
 -13*x*y^2 + x*y - 12*x*z^3 - 92*z
spoly(p,q,[x,y,z]);         yields
 x^3 - 13*x^2*y^2 - 12*x*y*z^3 + 1104*z^4
```
SEE ALSO : help(grobner), help(grobner,head)

9.5. The group package for permutation and finitely-presented groups

SYNTAX : group[<function>] (args); or <function> (args);

- To use a group function, type "with(group, <function>);" to define <function>, or type "with(group);" to define all group functions.
- The functions available are:

centralizer cosets cosrep groupmember grouporder
inter invperm isnormal mulperms normalizer
permrep pres

- In addition, convert(__,disjcyc), convert(__,permlist,__) and type(__,disjcyc) are defined.
- For example, to find the order of a permutation group:

```
with(group);
grouporder(permgroup(8, {a=[[1,2]], b=[[1,2,3,4,5,6,7,8]]}));
```

- For help with a particular function do: help(group, <function>);
- For help with how to represent groups, subgroups, and group elements for this package, see help(group,grelgroup), help(group,permgroup), and help(group,subgrel).

SEE ALSO : with, help(group,convert), help(group,type)

9.5.1. centralizer -- find the centralizer of a set of permutations in a permutation group

CALLING SEQUENCE : centralizer(<pg>, <s>);
PARAMETERS : <pg> – group in which the centralizer is to be found
<s> – set of permutations or a single permutation

- This function finds the largest subgroup of the permutation group <pg> in which every element commutes with every element of <s>. <s> need not be contained in <pg>. The result is returned as an unevaluated "permgroup" call.

EXAMPLES :

```
centralizer(permgroup(7, {[[1,2]], [[1,2,3,4,5,6,7]]}), {[[3,6]]});
```

yields:

```
  permgroup(7, {[[5, 7]], [[2, 4, 5]], [[1, 5, 2, 7, 4], [3, 6]]})
centralizer(permgroup(5, {[[1,2,3]], [[3,4,5]]}), [[1,2]]);
```

yields permgroup(5, {[[1, 2], [4, 5]], [[3, 4, 5]]})

SEE ALSO : help(group,permgroup)

9.5.2. convert/disjcyc – convert a permutation in list notation

or a word into disjoint cycle notation
convert/permlist – convert a permutation in disjoint cycle notation into list notation

CALLING SEQUENCE : 1. convert(<lperm>,disjcyc) or
2. convert(<word>,disjcyc,<pg>) or
3. convert(<perm>,permlist,<deg>)

PARAMETERS :
1. <lperm> – a permutation in list notation
2. <pg> – a "permgroup" with named generators
   <word> – a word in the generator names of <pg>
3. <perm> – the permutation
   <deg> – the degree of the permutation

1. The argument <lperm> is expected to be a permutation in list notation. This means that <lperm> is a list whose i'th element is the image of i under the permutation. The permutation is returned in disjoint cycle notation.
2. <word> is expected to be a word in the generator names of <pg>. In other words, <word> is a list whose elements are generator names and/or their inverses. The product represented by <word> is computed and returned in disjoint cycle notation.
3. A list is returned whose i'th element is the image of i under <perm>, for i = 1..<deg>.

EXAMPLES :

```
convert([3,4,1,2,7,6,5],disjcyc);
   yields [[1, 3], [2, 4], [5, 7]]
pg := permgroup(7, {a=[[1,2]], b = [[1,2,3,4,5,6,7]]}):
convert([a,b,1/a],disjcyc,pg);
   yields [[1, 3, 4, 5, 6, 7, 2]]
convert([[2,4,1], [7,3]],permlist,7);
   yields   [2, 4, 7, 1, 5, 6, 3]
```

SEE ALSO : help(group,permgroup), help(group,grelgroup)

9.5.3. cosets – find a complete list of right coset representatives

for a subgroup of a permutation group or a group given by generators and relations

CALLING SEQUENCE : cosets(<sbgrl>) or
cosets(<pg>, <sbpg>);

PARAMETERS : <sbgrl> – a subgroup of a group given by generators and relations
<pg>,<sbpg> – permutation groups of same degree

- For groups given by generators and relations, the argument <sbgrl> should be a "subgrel". A set of words in the generators of the group is returned.
- For permutation groups, both arguments should be "permgroup"s and <sg> should be a subgroup of <pg>. A set of permutations in disjoint cycle notation is returned.

EXAMPLES :
```
g := grelgroup({a,b,c}, {[a,b,c,a,1/b],[b,c,a,b,1/c],[c,a,b,c,1/a]}):
cosets(subgrel({y=[a,b,c]}, g));
yields   {[], [a], [a, b]}
pg1 := permgroup(7, {[[1,2]], [[1,2,3,4,5,6,7]]}):
pg2 := permgroup(7, {[[1,2,3]], [[3,4,5,6,7]]}):
cosets(pg1,pg2);
yields   {[[6, 7]], []}
```
SEE ALSO : help(group,permgroup), help(group,grelgroup), help(group,subgrel)

9.5.4. cosrep – express group element as product of an element in a subgroup multiplied by a right coset representative for that subgroup

CALLING SEQUENCE : cosrep(<elem>, <sub>)

PARAMETERS : <elem> – a permutation or a word in the group generators
<sub> – a permutation group or a "subgrel"

- If <sub> is a "subgrel", then <elem> should be a word in the group generators. A two-element list will be returned. The first element is the subgroup element expressed as a word in the subgroup generators, the second is the right coset representative. The coset representative will be an element of the set returned by "cosets(<sub>)".
- If <sub> is a "permgroup", then <elem> should be a permutation in disjoint cycle notation. A two-element list is returned. The first element is a permutation contained in <sub>, the second is a right coset representative permutation for <sub> in the symmetric group of the same degree. The coset representative will be an element of the set returned by "cosets(sn,<sub>)", where sn is the symmetric group of the same degree as <sub>.

EXAMPLES :
```
g := grelgroup({a,b,c}, {[a,b,c,a,1/b],[b,c,a,b,1/c],[c,a,b,c,1/a]}):
cosrep([c], subgrel({y=[a,b,c]}, g));
yields   [[y, y, y, y, y], [a]]
pg := permgroup(7, {[[1,2,3]], [[3,4,5,6,7]]}):
cosrep([[3,4,5,6]],pg);
yields   [[[3, 4, 5, 7, 6]], [[6, 7]]]
```
SEE ALSO : help(group,permgroup), help(group,subgrel), help(group,cosets)

9.5.5. grelgroup – represent a group by generators and relations

CALLING SEQUENCE : grelgroup(<gens>, <rels>);
PARAMETERS : <gens> – a set of names taken to be the generators of the group
<rels> – a set of relations among the generators which define the group

- The function "grelgroup" is used as a procedure and an unevaluated procedure call. As a procedure, "grelgroup" checks its arguments and then either exits with an error or returns the unevaluated "grelgroup" call.
- The first argument is a set of Maple names which stand for the generators of the group. The second argument is a set of "words" in the generators. A "word" is a list of generators and/or inverses of generators representing a product. The inverse of a generator g is represented by 1/g. An empty list represents the identity element. The words w1, w2, ... wn in <rels> are such that the relations w1 = w2 = ... = wn = 1 define the group.

EXAMPLES :
```
grelgroup({a,b}, {[a,a,a], [b,b], [a,b,1/a,1/b]});
  yields   grelgroup({a, b}, {[a, a, a], [b, b], [a, b, 1/a, 1/b]})
grelgroup({a,b}, {[a,1/c,a], [b,a]});
  yields   Error, (in grelgroup) invalid arguments
```
SEE ALSO : help(group,subgrel)

9.5.6. groupmember – determine whether a given permutation is in a given permutation group

CALLING SEQUENCE : groupmember(<perm>,<pg>)
PARAMETERS : <perm> – the permutation to test
<pg> – the permutation group

- true or false is returned, indicating whether <perm> is in <pg>.

EXAMPLES :

```
pg := permgroup(7, {[[1,2,3]], [[3,4,5,6,7]]}):
groupmember([[1,5],[3,6]], pg);          yields true
groupmember([[3,2,4,7]], pg);            yields false
```

SEE ALSO : help(group,permgroup)

9.5.7. grouporder – compute the order of a group

CALLING SEQUENCE : grouporder(<g>)
PARAMETERS : <g> – a "permgroup" or "grelgroup"

- The number of elements in the given group is computed and returned.

EXAMPLES :

```
grouporder(permgroup(7, {[[1,2,3]], [[3,4,5,6,7]]}));
  yields 2520
grouporder(grelgroup({x,y}, {[x,x,y,x,y,y,y],[y,y,x,y,x,x,x]}));
  yields 56
```

SEE ALSO : help(group,permgroup), help(group,grelgroup)

9.5.8. inter – find the intersection of two permutation groups

CALLING SEQUENCE : inter(<pg1>,<pg2>)
PARAMETERS : <pg1>,<pg2> – two "permgroup"s of same degree

- The intersection of two permutation groups is found, and returned in the form of a "permgroup".

EXAMPLES :

```
pg1 := permgroup(7, {[[2,3,4]], [[3,4,5,6,7]]}):
pg2 := permgroup(7, {[[1,2]], [[1,2,3,4,5,6]]}):
inter(pg1,pg2);
yields permgroup(7, {[[2, 4, 3]], [[2, 4, 6]], [[2, 4, 5]]})
```

SEE ALSO : help(group,permgroup)

9.5.9. invperm – find the inverse of a permutation given in disjoint cycle notation

CALLING SEQUENCE : invperm(<perm>)
PARAMETERS : <perm> – a permutation in disjoint cycle notation

- The inverse is computed, and the result is output in disjoint cycle notation.

EXAMPLES :
invperm([[4,6,5], [2,11]]); yields [[2, 11], [4, 5, 6]]
SEE ALSO : help(group,permgroup)

9.5.10. isnormal – determine whether a subgroup is normal

CALLING SEQUENCE : isnormal(<pg>, <sg>); or isnormal(<subgrl>);
PARAMETERS : <pg>,<sg> – two "permgroup"s of the same degree
<subgrl> – a "subgrel"

- With two arguments, this function determines whether <sg> is a normal subgroup of the permutation group generated by the union of <sg> and <pg>. If it is, then true is returned; otherwise false is returned.
- With one argument, this function tests whether <subgrl> is a normal subgroup. true or false is returned.

EXAMPLES :
```
pg := permgroup(8, {[[1,2]], [[1,2,3,4,5,6,7,8]]}):
sg1 := permgroup(8, {[[1,2,3,4]], [[1,2]], [[5,6,7,8]], [[5,6]]}):
isnormal(pg,sg1);            yields false
sg2 := permgroup(8, {[[1,2,3]], [[2,3,4,5,6,7,8]]}):
isnormal(pg,sg2);            yields true
g := grelgroup({a,b}, {[a,a,a,a,a], [b,b,b], [a,b,1/a,1/b]}):
isnormal(subgrel(x=[a], g)); yields true
```
SEE ALSO : help(group,permgroup), help(group,subgrel), help(group,grelgroup)

9.5.11. mulperms – multiply two permutations in disjoint cycle notation

CALLING SEQUENCE : mulperms(<perm1>,<perm2>)
PARAMETERS : <perm1>,<perm2> – the permutations in disjoint cycle notation

- The product is expressed in disjoint cycle notation.

EXAMPLES :
mulperms([[2,3,4], [1,6]], [[4,6]]); yields [[1, 4, 2, 3, 6]]
SEE ALSO : help(group,permgroup)

9.5.12. normalizer – find the normalizer of a subgroup of a permutation group

CALLING SEQUENCE : normalizer(<pg>, <sg>);
PARAMETERS : <pg>,<sg> – two "permgroup"s of the same degree

- This function finds the largest subgroup of <pg> in which <sg> is a normal subgroup. <sg> should be a subgroup of <pg>. The result is returned as an unevaluated "permgroup" call.

EXAMPLES :
```
pg := permgroup(7, {[[1,2,3]], [[3,4,5,6,7]]}):
sg := permgroup(7, {[[1,2,3]],[[3,4,5]]}):
normalizer(pg,sg);
```
yields permgroup(7, {[[1, 2, 3]], [[3, 4, 5]], [[4, 5], [6, 7]]})
SEE ALSO : help(group,permgroup)

9.5.13. permgroup – represent a permutation group

CALLING SEQUENCE : permgroup(<deg>, <gens>);
PARAMETERS : <deg> – degree of the permutation group
<gens> – set of generators for the permutation group

- The function "permgroup" is used as a procedure and an unevaluated procedure call. As a procedure, "permgroup" checks its arguments and then either exits with an error or returns the unevaluated "permgroup" call.
- The first argument is the degree of the group, and should be an integer. The second argument is a set of group generators. Each generator is represented in disjoint cycle notation. The generators may be named or unnamed. A named generator is an equation; the left operand is the generator's name, the right operand is the permutation in disjoint cycle notation.
- A permutation in disjoint cycle notation is a list of lists. Each sub-list represents a cycle; the permutation is the product of these cycles. The cycle [a[1], a[2], ..., a[n]] represents the permutation which maps a[1] to a[2], a[2] to a[3], ..., a[n–1] to a[n], and a[n] to a[1]. The identity element is represented by the empty list [].
- This package follows the convention that "permutations act on the right". In other words, if p1 and p2 are permutations, then the product of p1 and p2 (p1 &* p2) is defined such that (p1 &* p2)(i) = p2(p1(i)) for i=1..<deg>.

EXAMPLES :
```
permgroup(5, {a=[[1,2], [4,5]], b=[[5,4,3,2,1]]});
yields  permgroup(5, {a = [[1, 2], [4, 5]], b = [[5, 4, 3, 2, 1]]})
permgroup(6, {[[1,2]], [[1,2,3,4,5,6]]});
yields  permgroup(6, {[[1, 2]], [[1, 2, 3, 4, 5, 6]]})
permgroup(5, {x=[[3,4], y=[7,2]]});
yields  Error, (in permgroup) invalid parameters
```

9.5.14. permrep – find a permutation representation of a group given by generators and relations

CALLING SEQUENCE : permrep(<sbgrl>)
PARAMETERS : <sbgrl> – subgroup of a group given by generators and relations (i.e. a "subgrel")

- This function finds all the right cosets of the given subgroup in the given group, assigns integers consecutively to these cosets, constructs a permutation on these coset numbers for each group generator, and returns the permutation group generated by these permutations. Thus the permutation group will be a homomorphic image of (but not necessarily isomorphic to) the original group. A "permgroup" is returned whose generators are named the same as the original group generators.

EXAMPLES :
```
g := grelgroup({x,y}, {[x,x,y,x,y,y,y],[y,y,x,y,x,x,x]}):
sg := subgrel({y=[y]},g):
permrep(sg);
yields  permgroup(8, {y = [[2, 5, 8, 6, 7, 3, 4]],
                      x = [[1, 2, 3, 7, 5, 6, 4]]})
```
SEE ALSO : help(group,permgroup), help(group,grelgroup), help(group,subgrel)

9.5.15. pres – find a presentation for a subgroup of a group given by generators and relations

CALLING SEQUENCE : pres(<sbgrl>)
PARAMETERS : <sbgrl> – subgroup of a group given by generators and relations (i.e. a "subgrel")

- This function finds a set of relations among the given subgroup's generators sufficient to define the subgroup. The result is returned as a "grelgroup".

EXAMPLES :
```
g := grelgroup({a,b,c,d},
       {[a,b,c,1/d],[b,c,d,1/a],[c,d,a,1/b],[d,a,b,1/c]}):
sg := subgrel({x=[a,b],y=[a,c]},g):
pres(sg);
yields: grelgroup({x, y}, {[1/x, y, y, 1/x, 1/y, x, x, 1/y],
        [1/x, y, 1/x, 1/y, x, y, x, 1/y],
        [y, 1/x, 1/x, 1/x, y, x, 1/y, x, 1/y, x]})
```

SEE ALSO : help(group,grelgroup), help(group,subgrel)

9.5.16. subgrel – represent a subgroup of a group given by generators and relations

CALLING SEQUENCE : subgrel(<gens>, <grelgrp>);
PARAMETERS : <gens> – a set of equations designating the generators of the subgroup
<grelgrp> – the group for which this is a subgroup

- The function "subgrel" is used as a procedure and an unevaluated procedure call. As a procedure, "subgrel" checks its arguments and then either exits with an error or returns the unevaluated "subgrel" call.
- The first argument is a set of equations. The left side of each equation is the name of a subgroup generator; the right side is a word in the group generators (see grelgroup) representing the subgroup generator.

EXAMPLES :
```
subgrel({x=[a,b,1/a]}, grelgroup({a,b}, {[a,a,a], [b,b]}));
  yields   subgrel({x = [a, b, 1/a]},
     grelgroup({a, b}, {[a, a, a], [b, b]}))
subgrel({y=[a,b,c]}, grelgroup({a,b}, {[a,a], [b,a]}));
  yields   Error, (in subgrel) invalid arguments
```
SEE ALSO : help(group,grelgroup)

9.5.17. type/disjcyc – check permutation in disjoint cycle notation

CALLING SEQUENCE : type(<ex>,disjcyc,<deg>)
PARAMETERS : <ex> – the expression to check
<deg> – the intended degree of the permutation

- true or false is returned, depending on whether <ex> is a valid permutation of degree <deg> in disjoint cycle notation.

EXAMPLES :

```
type([1,2,3,4,5],disjcyc,6);          yields false
type([[1,2,4],[3,9,6]],disjcyc,7);    yields false
type([[1,2,4],[3,5,6]],disjcyc,7);    yields true
```

SEE ALSO : help(group,permgroup)

9.6. The `linalg` package for Linear Algebra

SYNTAX : linalg[<function>] (args); or <function> (args);

- To use a linalg function, type "with(linalg, <function>);" to define <function>, or type "with(linalg);" to define all linalg functions.
- The functions available are:

add	addcol	addrow	adj	adjoint	angle
band	bezout	coldim	colspace	cond	crossprod
curl	definite	det	diverge	dotprod	eigenvals
gausselim	genmatrix	grad	hadamard	hessian	hilbert
indexfunc	inverse	ismith	jacobian	kernel	laplacian
leastsqrs	linsolve	mdet	mulcol	mulrow	multiply
norm	nullspace	orthog	permanent	range	rank
rowdim	rowspace	scalarmul	singularvals	smith	submatrix
subvector	swapcol	swaprow	sylvester	toeplitz	trace
transpose	vandermonde	vectdim			

- For help with a particular function do: help(linalg, <function>);
- E.g., to multiply two matrices A and B, type:
  with(linalg,multiply); multiply(A,B);
- The package functions are always available without applying "with", using the long-form notation:
  linalg[<function>] (<args>);
  This notation is necessary whenever there is a conflict between a package function name and another function used in the same session.

SEE ALSO : evalm, with

9.6.1. add – matrix or vector addition

CALLING SEQUENCE : add(A,B), add(A,B,c1,c2)
PARAMETERS : A, B – matrices or vectors (with consistent dimensions)
c1, c2 – (optional) scalar parameters.

- add(A,B) computes the matrix (vector) sum of A and B
- add(A,B,c1,c2) computes the matrix (vector) sum c1*A + c2*B.
- The "evalm" command is an easy method of accessing "add".

EXAMPLES :

```
A := array( [ [1,2,3], [2,3,4], [3,4,5] ] );
I := array( 1..3, 1..3, identity );
add( A, I, 1, 10 );    yields
                       array( [ [11,2,3], [2,13,4], [3,4,15] ] );
```

SEE ALSO: evalm

9.6.2. addcol – form linear combinations of matrix columns

CALLING SEQUENCE : addcol(A,c1,c2,m);

PARAMETERS :

A	– matrix
m	– a (scalar) expression
c1, c2	– integers specifying columns of A

- "addcol" computes the matrix identical to A, except that the new matrix has the column c2 replaced by: m * column c1 + column c2.

EXAMPLES :

```
A := array( 1..3,1..3, [ [1,2,3], [2,3,4], [3,4,5] ] );
addcol(a, 1, 2, 10 );    yields
                         array( [ [1,12,3], [2,23,4], [2,34,5] ] ).
```

9.6.3. addrow – form linear combinations of matrix rows

CALLING SEQUENCE : addrow(A,r1,r2,m);

PARAMETERS :

A	– matrix
m	– a scalar quantity which will be a multiplier
r1, r2	– m times row r1 is added to row r2 in the result

- "addrow" computes the matrix identical to A, except that the new matrix has the row r2 replaced by: m * row r1 + row r2.

EXAMPLES :

```
a := array( 1..3,1..3, [ [1,2,3], [2,3,4], [3,4,5] ] );
addrow(a, 1, 2, 10 );    yields
                         array( [ [1,2,3], [12,23,34], [3,4,5] ] ).
```

9.6.4. adjoint – compute adjoint of a matrix

CALLING SEQUENCE : adjoint(A); or adj(A);

PARAMETERS : A – n x n matrix (with all elements assigned)

- "adjoint" computes the matrix such that the matrix product of A and the adjoint(A) is the product of det(A) and the matrix identity. This is done through the computation of minors of A.

9.6.5. angle – compute angle between vectors

CALLING SEQUENCE : angle(u,v)
PARAMETERS : u,v – n–dimensional vectors

- angle(u,v) computes the angle theta between the n–dimensional vectors u and v via the formula cos(theta) = sqrt(u.v / |u| |v|), using the Euclidean inner product and norm.

EXAMPLES :

```
angle( array([1,0]), array([0,1]) );   yields 1/2 Pi
angle( array([1,1]), array([0,1]) );   yields 1/4 Pi
angle( array([1,0]), array([1,0]) );   yields 0
```

9.6.6. band – create a band matrix

CALLING SEQUENCE : band(b,n)
PARAMETERS : n – positive integer.
b – list or vector with no more than n elements.

- band(b,n) creates an n x n banded matrix using the elements of b. The number of elements of b must be odd. All the elements of the i-th subdiagonal are initialized to b[i], for the first half of b. The elements of the main diagonal of the matrix result are initialized the middle element of b. The last half of b is to initialize the superdiagonals. If n is larger than the length of b, then the remaining sub– and super– diagonals are zero. The matrix result uses a sparse indexing function.

EXAMPLES :
band([1],3) yields a 3x3 identity matrix.
band([1,2,–1],4) yields a 4x4 matrix, with 1 on the main subdiagonal, –1 on main superdiagonal and 2 on main diagonal.

9.6.7. bezout – create the Bezout matrix from two polynomials

CALLING SEQUENCE : bezout(p,q,x)
PARAMETERS : x – a variable (name)
p, q – polynomials in the variable x

- bezout(p,q,x) returns the Bezout matrix of the polynomials p and q with respect to x. The determinant of this matrix equals resultant(p,q,x).
- The polynomials p and q must be expanded because the coeff function is used on p and q to determine the entries of the matrix.

EXAMPLES :

```
p := a+b*x+c*x^2:
q := d+e*x+f*x^2:
bezout(p,q,x);    yields  array( sparse, 1..2,1..2,
     [ [d*c-a*f, d*b-a*e], [e*c-b*f, d*c-a*f] ] )
```

9.6.8. coldim – determine the column dimension of a matrix

CALLING SEQUENCE : coldim(A)
PARAMETERS : A – a matrix

- coldim(A) returns an integer, the number of columns in the matrix A.

EXAMPLES :

```
coldim( array(1..2,1..3) );    yields  3
```

9.6.9. colspace or range – compute a basis for the column space

CALLING SEQUENCE : colspace(A) or colspace(A,'dim'), or range(A) or range(A,'dim')
PARAMETERS : A – an n x m matrix
dim – (optional) parameter which takes on the dimension of the column space

- "range" and "colspace" are synonyms.
- Gaussian elimination is performed on the transpose of A. Vectors corresponding to the non-zero rows of the result are returned in a set. If present, the variable dim is assigned the dimensionality of the column space, that is, the number of vectors in the set returned. An error results if not all the elements of A have been assigned values.

EXAMPLES :

```
A := array( [ [1,2,3],[1,3,5],[2,5,8] ] );
colspace( A );    yields  {array([1,1,2]),array([0,1,1])}
Note: Gaussian elimination applied to the transpose of A yields:
                array( [ [1,1,2],[0,1,1],[0,0,0] ] )
```

9.6.10. cond – condition number of a matrix

CALLING SEQUENCE : cond(A) or cond(A,normname);
PARAMETERS : A – a square matrix
normname – (optional) matrix norm, must be one of: 1, 2, 'infinity', or 'frobenius'.

- cond(A) computes the "standard" matrix condition number, defined as norm(A) * norm(inverse(A)). The matrix norm is the default employed by the linalg[norm] function, namely the infinity norm (maximum row sum).
- cond(A,normname) computes
  norm(A,normname) * norm(inverse(A),normname) .
  This is the same measure, but using the specified norm instead of the infinity norm.

EXAMPLES :

```
I := array(1..2,1..2, identity);
cond(I);        yields  1
A := array([[1,2],[3,4]]);
cond(A,1);      yields  21
                    I.e., norm(A,1)*norm(inverse(A),1)
```

9.6.11. crossprod – vector cross product

CALLING SEQUENCE : crossprod(u,v)
PARAMETERS : u, v – lists or vectors, each with 3 elements

- Computes the vector cross-product of u and v, defined to be
  array([u[2]*v[3]–u[3]*v[2], u[3]*v[1]–u[1]*v[3], u[1]*v[2]–u[2]*v[1]])

9.6.12. curl – curl of a vector

CALLING SEQUENCE : curl(f,v)
PARAMETERS : f – a list or vector of three expressions
v – a list or vector of three variables

- The curl of f with respect to v is computed. The curl is defined to be a vector of length 3:

```
[ diff(f[3],v[2]) – diff(f[2],v[3]),
  diff(f[1],v[3]) – diff(f[3],v[1]),
  diff(f[2],v[1]) – diff(f[1],v[2]) ];
```

  An error results if the arguments are not of length three, or if the elements of v are not valid symbols to be differentiated with respect to.

EXAMPLES :

```
f := [ x^2, x*z, y^2*z ];
v := [ x, y, z ];
curl(f, v);    yields   [ 2*y*z–x, 0, z ]
```

9.6.13. definite – test for positive (negative) definite matrices

CALLING SEQUENCE : definite(A, kind)
PARAMETERS : A – a square symmetric matrix
kind – one of the symbols: positive_def, positive_semidef, negative_def, or negative_semidef

- For numerical matrices, "definite" returns true if the matrix has the property specified by the parameter "kind", i.e. positive definite, positive semidefinite, negative definite, or negative semidefinite.
- For matrices with non-numerical entries, "definite" returns a conjunction of boolean expressions, all of which must be true if the matrix is to have the property specified by the parameter "kind".

EXAMPLES :

```
A := array( 1..2, 1..2, symmetric, [[2,1],[1,3]] ):
definite( A,'positive_def' );          yields  true
B := array( 1..2, 1..2, symmetric ):
definite( B,'negative_semidef' );   yields
        B[1,1]<=0  and -B[1,1]*B[2,2] + B[1,2]^2<=0.
```

9.6.14. det – determinant of a matrix

CALLING SEQUENCE : det(A), det(A, sparse)
PARAMETERS : A – a two-dimensional square array
sparse – optional directive

- The function "det" must first be defined via "with(linalg,det)" or "with(linalg)".
- The 2nd argument (optional) specifies that the computational method of minor expansion should be used. This works well for sparse matrices.
- With no extra argument, a decision is made automatically whether to use minor expansion or Gaussian elimination (or a combination of both). Specifying the extra argument eliminates the overhead of the decision process if you happen to know ahead of time that minor expansion will work best.

EXAMPLES :

```
A := array( 1..3, 1..3, [[1,1/2,1/3],[1/2,1/3,1/4],[1/3,1/4,1/5]] );
det( A, sparse );   yields  1/2160
```

9.6.15. diverge – compute the divergence of a vector

CALLING SEQUENCE : diverge(f, v)
PARAMETERS : f – a vector or list of expressions
v – a vector or list of the same length as f

- diverge computes the divergence of f with respect to v, defined as the sum of diff(f[i],v[i]) as i ranges over the length of f.

EXAMPLES :

```
f := array( [x,y^2,z] );
v := array( [x,y,z] );
diverge( f,v );    yields   2+2*y
```

9.6.16. dotprod – vector dot (scalar) product

CALLING SEQUENCE : dotprod(u,v)
PARAMETERS : u,v – lists or vectors of the same length

- dotprod computes the vector dotproduct, defined to be sum u[i]*v[i], as i ranges over the length of u and v.

EXAMPLES :

```
u := array( [1,x,y] );
v := array( [1,0,0] );
dotprod(u,v);    yields   1
```

9.6.17. eigenvals – compute the eigenvalues of a matrix

CALLING SEQUENCE : eigenvals(A) or eigenvals(A,C)
PARAMETERS : A – a square matrix
C – an optional matrix of the same shape as A

- eigenvals(A) returns a sequence of the eigenvalues of A. The eigenvalues are computed by solving the characteristic polynomial det(A – lambda I) for the scalar variable lambda, where I is the identity matrix. If the characteristic polynomial is of degree greater than five, then it may not be possible to find all the eigenvalues.
- eigenvals(A,C) solves the "generalized eigenvalue problem" – finding the roots of the polynomial det(A – lambda C).
- If the last argument of the function call is the name "implicit", eigenvalues which are roots of irreducible polynomials of degree 2, 3 or 4 will be expressed as RootOfs.

EXAMPLES :

```
A := array( [[1,2,3],[1,2,3],[1,5,6]] );
eigenvals(A);      yields  0, 0, 9
B := array(1..2,1..2,diagonal);
eigenvals(B,B);    yields  1, 1
```

9.6.18. gausselim – fraction-free Gaussian elimination on a matrix

CALLING SEQUENCE : gausselim(A, 'rank', 'piv', 'sgn');

PARAMETERS :
- A – a matrix
- 'rank' – for returning rank of A (optional)
- 'piv' – for returning pivot vector (optional)
- 'sgn' – for returning sign of permutation (optional)

- Fraction-free Gaussian elimination with row pivoting is performed on A. It is assumed that the matrix coefficients are polynomials over the rationals. The matrix result of fraction-free Gaussian elimination is returned (a row permutation of an upper triangular matrix).
- The second parameter is used to return the number of non-zero rows in the result. The third parameter is a vector containing the integers 1..rowdim(A) permuted to indicate the row interchanges as a result of pivoting. The fourth parameter is assigned +1 or −1 depending on whether there were an even or odd number of row interchanges during the elimination.
- Note that if n is the dimension of A, then the determinant of A is available from the matrix result G of gausselim as:

$$\text{sgn} * G[\text{piv}[n],n] .$$

EXAMPLE:

```
A := array([[x,1,0],[0,0,1],[1,y,1]]);
gausselim( A,'rank','piv','sgn' ); yields
            array( [[x, 1, 0], [0, 0, y x − 1], [0, y x − 1, x]] )
rank;       yields  3
op(piv);    yields  array( [1, 3, 2] )
sgn;        yields  −1
```

9.6.19. genmatrix – generate the coefficient matrix from equations

CALLING SEQUENCE : genmatrix(eqns,vars) or genmatrix(eqns,vars,flag)

PARAMETERS :
- eqns, vars – sets or lists of equations and variables
- flag – (optional) a flag which can be any value

- genmatrix generates the coefficient matrix from the linear system of equations "eqns" in the unknowns "vars".
- If an optional third argument is present, the negative of the "right hand side" vector will be included as the last column of the matrix.

EXAMPLES :

```
genmatrix( [x+2*y-3,3*x-5*y=0], [x,y] );
                array( [ [1,2], [3,-5] ] )
genmatrix( [x+2*y-3,3*x-5*y=0], [x,y], flag );
              array( [ [1,2,-3], [3,-5,0] ] )
```

9.6.20. grad – vector gradient of an expression

CALLING SEQUENCE : grad(<expr>, v)
PARAMETERS : <expr> – scalar expression
v – vector or list of variables

- grad computes the gradient of expr with respect to v. That is, it computes the vector of partial derivatives:
  array([diff(expr,v[1]), diff(expr,v[2]), ...]) .

EXAMPLES :

```
grad( x*y*z,[x,y,z] );          yields  array( [y*z,x*z,x*y] )
grad( x*y*z,array([x,y,z]) );   yields  array( [y*z,x*z,x*y] )
```

9.6.21. hadamard – bound on coefficients of the determinant of a matrix

CALLING SEQUENCE : linalg[hadamard](A) or hadamard(A)
PARAMETERS : A – a square matrix

- Compute a bound on the maxnorm of det(A) where A is an n by n matrix whose coefficients lie in the domain Z, Q, R, Z[x], Q[x] or R[x].
- For Z, the classical inequality of Hadamard is :
  |det(A)| <= sqrt(product(sum(A[i,j]^2,j=1..n), i=1..n))

9.6.22. hessian – compute the Hessian matrix of an expression

CALLING SEQUENCE : hessian(expr, vars)
PARAMETERS : expr – a scalar expression
vars – a vector or list of variables

- hessian(expr,vars) computes the hessian matrix of expr with respect to vars. The matrix result is n x n, where n is the length of vars. The (i,j)th entry of the matrix result is diff(expr,vars[i],vars[j]).

EXAMPLES :

hessian(x*y*z, [x,y,z]) yields:
array(1..3, 1..3, [[0,z,y], [z,0,x], [y,x,0]])

9.6.23. hilbert – create a Hilbert matrix

CALLING SEQUENCE : hilbert(n) or hilbert(n,x)
PARAMETERS : n – a positive integer
x – optional expression

- hilbert(n,x) returns the n by n generalized Hilbert matrix. This matrix is symmetric and has 1/(i+j–x) as its (i,j)-th entry. If x is not specified, the Hilbert matrix has as its (i,j)-th entry 1/(i+j–1).

EXAMPLES :

hilbert(3); yields:
array(symmetric, 1..3,1..3, [[1,1/2,1/3], [1/2,1/3,1/4], [1/3,1/4,1/5]]);

9.6.24. indexfunc – determine the indexing function of an array

CALLING SEQUENCE : indexfunc(A);
PARAMETERS : A – an array

- indexfunc(A) returns the indexing function for the array A. This might be one of the standard types such as: 'identity', 'symmetric', or 'diagonal', or the name of a user-defined indexing function, or NULL if there is no indexing function specified.

EXAMPLES :

indexfunc(array(1..2,1..2,symmetric)); yields symmetric
indexfunc(array(1..2,1..2,[[1,0],[0,1]])); yields NULL

9.6.25. inverse – compute the inverse of a matrix

CALLING SEQUENCE : inverse(A)
PARAMETERS : A – a square matrix

- inverse(A) computes the matrix inverse of A. An error occurs if the matrix is singular.
- Inverse uses Cramer's rule for matrices of dimension less than or equal to 4 by 4, and matrices with the 'sparse' indexing function. For other matrices, the inverse is computed by applying the operations for the

Gauss-Jordan reduction of A to an identity matrix of the same shape.

EXAMPLES :
```
A := array( [[1,x],[2,3]] );
inverse( A ); yields:
array(1..2,1..2, [-3/(-3+2*x),x/(-3+2*x)], [2/(-3+2*x),-1/(-3+2*x)])
```

9.6.26. ismith

CALLING SEQUENCE : ismith(A);
PARAMETERS : A – a rectangular matrix of integers

- Computes the smith normal form of an n by m rectangular matrix of integers. If two n by n matrices have the same smith normal form, they are equivalent. The smith normal form is a diagonal matrix where:
  sign(A[i,i]) = 1 for 0 < i <= min(n,m)
  A[i,i] divides A[i+1,i+1] for 0 < i < min(n,m)
  |det(A)| = product(A[i,i],i=1..n) if n = m
- The smith normal form is obtained by doing elementary row and column operations. This includes interchanging a row (column), multiplying through a row (column) by –1, and subtracting an integral multiple of one row (column) from another.

EXAMPLE :
```
H := array( [[9,-36,30], [-36,192,-180], [30,-180,180]] ):
linalg[ismith](H);  yields
   array( 1..3, 1..3, [[3,0,0], [0,12,0], [0,0,60]] )
```
SEE ALSO: smith

9.6.27. jacobian – compute the Jacobian matrix of a vector function

CALLING SEQUENCE : jacobian(f,v)
PARAMETERS : f – a vector or list of expressions
v – a vector or list of variables

- jacobian(f,v) computes the Jacobian matrix of f with respect to v. The (i,j)-th entry of the matrix result is diff(f[i],v[j]).

EXAMPLES :
```
A := array( [x^2, x*y, x*z] );
jacobian( A,[x,y,z] );  yields:
array(1..3,1..3,[ [2*x,0,0], [y,x,0], [z,0,x] ])
```

9.6.28. kernel or nullspace – compute a basis for the null space

CALLING SEQUENCE : kernel(A,'nulldim') or
nullspace(A,'nulldim')

PARAMETERS : A – a matrix
'nulldim' – an optional unevaluated name

- "nullspace" and "kernel" are synonyms.
- kernel(A) computes a basis for the null space (kernel) of the linear transformation defined by the matrix A. The result is a set of vectors, possibly empty.
- kernel(A,'nulldim') in addition to the above, assigns the variable nulldim the vector space dimension of the kernel (a non-negative integer).

EXAMPLES :

```
A := array( [[1,2,3],[1,x,3],[0,0,0]] );
kernel( A );   yields  { array( 1..3, [-3, 0, 1]) }
```

9.6.29. laplacian – compute the Laplacian of an expression

CALLING SEQUENCE : laplacian(expr, v)

PARAMETERS : expr – a scalar expression
v – a vector or list of variables

- laplacian(expr,v) computes the laplacian of expr with respect to v. This is defined to be the sum of the second derivatives diff(expr,v[i]$2), with v[i] ranging over the components of v.

EXAMPLES :

```
laplacian(x^2*y*z,[x,y,z]);   yields  2 y z
```

9.6.30. leastsqrs – least-squares solution of equations

CALLING SEQUENCE : leastsqrs(A,b) or leastsqrs(s1,s2)

PARAMETERS : A – a matrix
b – a vector
s1 – a set of expressions
s2 – a set of variables

- leastsqrs(A,b) returns the vector that best satisfies A x = b in the least-squares sense. I.e., the result returned is the vector which minimizes norm(Ax–b,2) when all vectors x are considered.

- leastsqrs(s1,s2) finds the values for the variables in s2 which minimize the expressions in s1 in the least-squares sense. The result returned is a set of equations of the form
  {var1=expr1, var2=expr2 ,..., varn=exprn}.

EXAMPLES :

```
A := array( [[1,1,1],[1,2,4],[1,0,0],[1,-1,1]] );
b := array( [3,10,3,9/10] );
leastsqrs(A,b);          yields  array( [327/200,301/200,49/40] )
S := {c0+c1+c2-3, c0+2*c1+4*c2-10, c0-9/10, c0-c1+c2-3};
leastsqrs(S,{c0,c1,c2});  yields  {c0=159/200,c1=7/200,c2=91/40}
```

9.6.31. linsolve – solution of linear equations

CALLING SEQUENCE : linsolve(A,b)
PARAMETERS : A – a matrix
b – a vector

- linsolve(A,b) finds the vector x which satisfies the matrix equation A x = b. If A has n rows and m columns, then vectdim(b) must be n. vectdim(x) will be m, if a solution exists. If the equation A x = b has no solution, then the null sequence NULL is returned. If the system of equations has many solutions, then the result will use the symbols t1, t2, t3, ... to describe the family of solutions parametrically.
- linsolve assumes that the entries of A and b are polynomials over the rational numbers.

EXAMPLES :

```
A := array([[1,2],[1,3]]);
V := array([1,2]);
linsolve( A,V );   yields  array( 1..2, [-1,1] )
```

9.6.32. mdet – determinant of a matrix mod p

CALLING SEQUENCE : mdet(A,p)
PARAMETERS : A – a two-dimensional square array

- The function "mdet" computes the determinant of a matrix modulo p . The two domains Zp and Zp[x] are understood.
- The function "mdet" must first be defined via "with(linalg,mdet)" or "with(linalg)".

9.6.33. mulcol – multiply a column of a matrix by an expression

CALLING SEQUENCE : mulcol(A, c, expr)
PARAMETERS : A – a matrix
c – a positive integer
<expr> – a scalar expression

- mulcol(A,c,<expr>) returns a matrix which has the same entries as A, except that each entry in column c is multiplied by <expr>.
- An attempt is made to preserve particular indexing structure (e.g. identity, diagonal, sparse) as best it can.

EXAMPLES :
```
mulcol( array([[1,2],[3,4]]),2,2 );   yields  array([[1,4],[3,8]])
A := array(sparse, 1..2,1..2, [[1]]);
mulcol( A,1,2 );                      yields  array(sparse,1..2,1..2,[[2]])
```

9.6.34. mulrow – multiply a row of a matrix by an expression

CALLING SEQUENCE : mulrow(A, r, expr)
PARAMETERS : A – a matrix
r – a positive integer
<expr> – a scalar expression

- mulrow(A,r,<expr>) returns a matrix which has the same entries as A, except that each entry in row r is multiplied by <expr>.
- An attempt is made to preserve particular indexing structure (e.g. identity, diagonal, sparse) as best it can.

EXAMPLES :
```
mulrow( array([[1,2],[3,4]]),2,2 );   yields  array([[1,2],[6,8]])
A := array(sparse, 1..2,1..2, [[1]] );
mulrow( A,1,2 );                      yields  array(sparse,1..2,1..2,[[2]])
```

9.6.35. multiply – matrix-matrix or matrix-vector multiplication

CALLING SEQUENCE : multiply(A,B) or multiply(A,v)
PARAMETERS : A – a matrix
B – a matrix
v – a vector or list

- multiply(A,B) calculates the matrix product A B. The definition of matrix multiplication requires that the number of columns of A must be the same as the number of rows of B. If A is an n x m matrix, and B a m x k matrix, the result is a matrix with n rows and k columns.

- multiply(A,v) calculates the matrix-vector product A v. The number of entries in v must be equal to the number of columns of A. Thus if A is an n x m matrix, vectdim(v) must be m. The result is a vector with n entries.
- The "evalm" command provides an easy method of accessing "multiply".

EXAMPLES :

```
A := array( [[1,2],[3,4]] );
B := array( [[0,1],[1,0]] );
multiply(B,A);      yields  array([[3,4],[1,2]])
multiply(A,[1,2]);  yields  array([5,11])
```

SEE ALSO: evalm

9.6.36. norm – norm of a matrix

CALLING SEQUENCE : norm(A, normname) or norm(v, normname)

PARAMETERS :

A	– a matrix
v	– a vector
normname	– a matrix/vector norm For matrices, one of: 1, 2, infinity, frobenius For vectors, one of: any positive integer, infinity, frobenius

- norm(A, normname) computes the specified matrix norm for the matrix A.
- norm(v, normname) computes the specified vector norm for the vector v.

EXAMPLES :

```
norm( array([[1,–2],[3,–4]]), infinity );   yields  7
norm( array([1,–1,2]), 2 );                 yields  6^(1/2).
```

9.6.37. orthog – test for orthogonal matrices

CALLING SEQUENCE : orthog(A)

PARAMETERS : A – a matrix

- orthog(A) returns true if the matrix A is orthogonal (the inner product of any column of A with itself is 1, and the inner product of any column of A with any other column is 0), false otherwise.
- The Maple function normal is used to simplify the expressions; if the entries of A are not pure rational expressions then it cannot be guaranteed that expressions equivalent to 0 or 1 are recognized.

- The polynomials p and q must be expanded in x, because the coeff function is used on p and q to determine the entries of the matrix (see "coeff").

EXAMPLES :

```
p := a+b*x:
q := c+d*x+e*x^2:
sylvester(p,q,x);    yields
array( sparse, 1..3,1..3, [[b,a,0], [0,b,a], [e,d,c]] )
```

9.6.50. toeplitz – create a Toeplitz matrix

CALLING SEQUENCE : toeplitz(L)
PARAMETERS : L – a list of expressions

- toeplitz(L) returns the symmetric toeplitz matrix corresponding to the list L.
- The matrix result has as many rows (and columns) as there are elements in the list L. The first element of L is placed all along the diagonal, and the i-th element of L is placed all along the (i–1)-th sub– and super– diagonals of the result.

EXAMPLES :

```
toeplitz( [a,b,c] );  yields:
  array( 1..3,1..3,symmetric, [[a,b,c],[b,a,b],[c,b,a]] )
```

9.6.51. trace – the trace of a matrix

CALLING SEQUENCE : trace(A)
PARAMETERS : A – a matrix

- trace(A) computes the trace of the matrix A. The trace of a matrix is defined to be the sum of the diagonal elements of A.

EXAMPLES :

```
trace( array([[1,2],[1,4]]) );    yields  5
```

9.6.52. transpose – compute the transpose of a matrix

CALLING SEQUENCE : transpose(A);
PARAMETERS : A – an m x n matrix

- The matrix transpose of A is computed. The result is an n x m matrix. The [i,j]-th element of the result is equal to the [j,i]-th element of A. The result inherits the indexing (e.g. diagonal, sparse) function of A, if it has one.

EXAMPLES :
```
rowspace(array([[0,0,0,1,0],[0,x,0,0,0],[0,0,y,y,0],[0,x,0,1,0]]),'d');
yields:
  {array(1..5, [0, 0, 0, y x, 0]), array(1..5, [0, 0, y x, y x, 0]),
   array(1..5, [0, x, 0, 0, 0])}
d;   yields  3
```

9.6.42. scalarmul – multiplication of a matrix/vector by an expression

CALLING SEQUENCE : scalarmul(A,expr) or scalarmul(v,expr)
PARAMETERS : A – a matrix
v – a vector
expr – a scalar expression

- scalarmul(A,expr) returns the matrix which is the result of multiplying A by the scalar value expr.
- scalarmul(v,expr) returns the result of multiplying the vector v by the scalar expr.

EXAMPLES :
```
scalarmul(array([[1,2],[3,4]]),2);   yields  array([[2,4],[6,8]])
scalarmul(array([1,2,3,4]),3);       yields  array([3,6,9,12])
```

9.6.43. singularvals – compute the singular values of a matrix

CALLING SEQUENCE : singularvals(A)
PARAMETER : A – a square matrix

- singularvals(A) returns a list of the singular values of A. These are the square roots of the eigenvalues of A transpose * A.

EXAMPLE :
```
A := array( [[1,0,1],[1,0,1],[0,1,0]] );
singularvals(A);   yields  [1,2,3]
```
SEE ALSO : linalg/eigenvals

9.6.44. smith

CALLING SEQUENCE : smith(A); or smith(A,x);
PARAMETERS : A – square matrix of polynomial entries
x – variable name used for calls to divide

- Computes the Smith canonical form of a matrix with polynomial entries. If only one indeterminate occurs in A, then only one argument need be supplied. In all other cases, the user must provide a second argument indicating a specific indeterminate x. The polynomials will

EXAMPLES :
orthog(array([[−1/2,sqrt(3)/2],[sqrt(3)/2,1/2]])); yields true

9.6.38. permanent – compute the permanent of a matrix

CALLING SEQUENCE : permanent(A)
PARAMETER : A – a square matrix

- The function permanent(A) returns the permanent of A.

EXAMPLE :

```
A := array( [[1,2,3],[4,5,6],[7,8,9]] );
permanent(A);   yields 450
```

9.6.39. rank – rank of a matrix

CALLING SEQUENCE : rank(A)
PARAMETERS : A – a matrix

- rank(A) computes the rank of a matrix, by performing Gaussian elimination on the rows of A.

EXAMPLES :
rank(array([[x,1,0],[0,0,1],[x*y,y,1]])); yields 2

9.6.40. rowdim – determine the row dimension of a matrix

CALLING SEQUENCE : rowdim(A)
PARAMETERS : A – a matrix

- rowdim(A) returns an integer, the number of rows in the matrix A.

EXAMPLES :
rowdim(array(1..2,1..3)); yields 2

9.6.41. rowspace – compute a basis for the row space of a matrix

CALLING SEQUENCE : rowspace(A) or rowspace(A,'dim')
PARAMETERS : A – a matrix
'dim' – an optional unevaluated name

- rowspace(A) returns a set of vectors, a basis for the vector space spanned by the vectors which are the rows of A. The optional second parameter dim is assigned an integer, the number of vectors returned.

- Note, either the row or column entry must be a single number specifying the desired row or column.

EXAMPLES :

```
A := array( [[1,2,3],[4,x,6]] );
subvector(A,1..2,2);    yields  array( [2,x] )
subvector(A,2,[2,1]);   yields  array( [x,4] )
```

SEE ALSO: help(linalg,submatrix)

9.6.47. swapcol – swap two columns in a matrix

CALLING SEQUENCE : swapcol(A,c1,c2)

PARAMETERS : A – a matrix
c1, c2 – positive integers

- swapcol(A,c1,c2) has as its result a matrix whose entries are the same as those of A, except that the entries in columns c1 and c2 are interchanged.

EXAMPLES :

```
A := array( [[1,2,x],[3,4,y]] );
swapcol( A,1,2 );   yields  array( [[2,1,x],[4,3,y]] )
```

9.6.48. swaprow – swap two rows in a matrix

CALLING SEQUENCE : swaprow(A,r1,r2)

PARAMETERS : A – a matrix
r1, r2 – positive integers

- swaprow(A,r1,r2) has as its result a matrix whose entries are the same as those of A, except that the entries in rows r1 and r2 are interchanged.

EXAMPLES :

```
A := array( [[1,2,x],[3,4,y]] );
swaprow( A,1,2 );   yields  array( [[3,4,y],[1,2,x]] )
```

9.6.49. sylvester – create Sylvester matrix from two polynomials

CALLING SEQUENCE : sylvester(p,q,x)

PARAMETERS : x – a variable (name)
p, q – expanded polynomials in the variable x

- sylvester(p,q,x) returns the Sylvester matrix of the polynomials p and q with respect to x.

then be regarded as elements of the ring R[x].

- The routine convert(... , characteristic) can be used to compute the characteristic matrix. When smith() is applied to such a matrix, the result is the rational canonical form of the original matrix.

EXAMPLES :
```
smith(array([[1-x,e],[0,1-x]]),x);  ==>
      array([ [1, 0], [0, 1-2*x+x^2] ])
convert(array([[t,u],[v,w]]), characteristic);  ==>
      array([ [_X-t, -u], [-v, _X-w] ])
smith(",_X);  ==>
      array([ [1, 0], [0, _X^2-_X*t-w*_X+w*t-v*u] ])
```

9.6.45. submatrix – extract a specified submatrix from a matrix

CALLING SEQUENCE : submatrix(A,Rrange,Crange); submatrix(A,Rlist,Clist)
PARAMETERS :

A	– a matrix
Rrange, Crange	– integer ranges of rows/columns of A
Rlist,Clist	– lists of integer row/column indices

- submatrix(A,Rrange,Crange) returns the submatrix of A selected by the row range Rrange and the column range Crange.
- submatrix(A,Rlist,Clist) returns the matrix whose (i,j)-th element is A[Rlist[i],Clist[j]].
- An optional fourth argument may be used to specify an indexing function for the resulting matrix.

EXAMPLES :
```
A := array( [[1,2,3],[4,x,6]] );
submatrix(A,1..2,2..3);    yields  array( [[2,3],[x,6]] )
submatrix(A,[2,1],[2,1]);  yields  array( [[x,4],[2,1]] )
```
SEE ALSO: help(linalg,subvector)

9.6.46. subvector – extract a specified vector from a matrix

CALLING SEQUENCE : subvector(A,row(s),col(s))
PARAMETERS :

A	– a matrix
row(s)	– can be a list, range, or a single value
col(s)	– can be a list, range, or a single value

- subvector(A,row(s),col(s)) returns the vector of A selected by the single value in either the row or col entry and the range or list in the other entry.

- If A is a vector, then it is treated as if it were a column (n x 1) matrix.

EXAMPLES :

```
A := array([[1,2,3],[4,5]]);
transpose(A);    yields  array([ [1,4],[2,5],[3,A[2,3]] ])
B := array(diagonal,1..2,1..2,[(1,1)=5]);
transpose(B);    yields  array(diagonal, [(1,1)=5,(2,2)=B[2,2]] )
C := array([1,2,3]);
transpose(C);    yields  array( 1..1,1..3, [[1,2,3]] );
```

9.6.53. vandermonde – create a Vandermonde matrix

CALLING SEQUENCE : vandermonde(L)
PARAMETERS : L – a list of expressions

- vandermonde(L) returns the Vandermonde matrix formed from the elements of the list. This square matrix has as its (i,j)-th entry L[i]^(j–1).

EXAMPLES :

```
vandermonde([1,2,3]);    yields  array( [[1,1,1],[1,2,4],[1,3,9]] )
```

9.6.54. vectdim – determine the dimension of a vector

CALLING SEQUENCE : vectdim(v) or vectdim(L)
PARAMETERS : v – a vector
L – a list

- vectdim(v) returns the dimension (number of elements) in the vector v. vectdim(L) returns the number of elements in L.

EXAMPLES :

```
vectdim( array([x,y,z]) );    yields  3
```

9.7. The np package for the Newman-Penrose Formalism

SYNTAX: np[<function>] (args); , or <function> (args);

- To use an np function, type "with(np, <function>);" to define <function>, or type "with(np);" to define all np functions.
- The functions available are:

  D V X Y (Pfaffian operators)

  V_D X_D Y_D X_V Y_V Y_X (commutators – SEE help(np,commutators);)

  conj (complex conjugation)

  eqns (initialization and display of NP equations)

  suball (a crude substitution utility)

- For help with a particular function do: help(np, <function>);
- For example, to apply the Pfaffian operator D to both sides of an equation "eq" , type:

  with(np,D); D(eq);

WARNING: This package uses global names for the various NP quantities; it is essential to be wary of unintentional conflicts.

9.7.1. Newman-Penrose Pfaffian operators D, V, X, Y

- The functions D, V, X, Y correspond to the Pfaffians associated with the null tetrad vectors l, n, m, m bar, respectively. They are simply applied as functions, and behave as differential operators on such expressions as are commonly encountered in the formalism (e.g. polynomials and rational functions in the NP quantities).
- These functions use a global variable "const" (initially {I}), which allows the marking of special names as constants.

EXAMPLES :

```
with(np):
eq1 := D(r) = r^2 + (e + ec)*p:
V( eq1 );                                              yields
  V(D(r)) = 2*r*V(r) + (e+ec)*V(p) + (V(e)+V(ec))*p
const := const union {L}: X( s^2 + R01 + L );    yields
  2*s*X(s) + X( R01 )
```

9.7.2. Newman-Penrose commutators V_D, X_D, Y_D, X_V, Y_V, Y_X

- The functions V_D, X_D, Y_D, X_V, Y_V, Y_X correspond to the commutation relations associated with the Lie brackets [V,D], [X,D], etc.. They return an equation whose left hand side is the Lie bracket, and whose right hand side is its known linear expansion in terms of the NP derivatives.

EXAMPLES :
```
with(np):
X_D( p );    yields
  X(D(p)) - D(X(p)) = (ac+b-pc)*D(p) + k*V(p)
                    - (rc+e-ec)*X(p) - s*Y(p)
```

9.7.3. conj – Newman-Penrose complex conjugation operator

- The function conj computes the complex conjugate of an object in the NP formalism according to the usual rules. The value of "conj(expr)" is output as "exprc" for readability; it is therefore important to realize that it will thereafter be treated as an independent quantity (except by the conj function!). The (Hermitian) Ricci spinor is treated accordingly.
- In addition, the conj function uses the sets real_fcns, imag_fcns to identify real and imaginary quantities (e.g. the operators D, V, and the constant I).

EXAMPLES :
```
with(np):
eq1 := D(r) = r^2 + (e + ec)*p:
conj( eq1 );                                          yields
  D(rc) = rc^2 + (ec+e)*pc
real_fcns := real_fcns union {W2}: conj( s^2 + R01 + W2 );    yields
  sc^2 + R10 + W2
```

9.7.4. eqns – initialization and display of NP equations

- The function eqns is used to initialize and/or display the set of "basic" equations, i.e. those named eq.k where 1 <= k <= neqs. An initial call of the form "eqns();" causes the equations eq1 through eq29 to be assigned (but not printed), and the variable neqs to be initialized to 29.
- Once the equations have been initialized, a sequential subset of them is printed by typing "eqns(a,b);" (to display eq.a to eq.b), or "eqns();" to display eq1 to eq.neqs (whatever the current value of neqs). Note that

the user must reset "neqs" when the set of basic equations is extended.

EXAMPLES : (equations already initialized via "with(np): eqns();")

```
k := 0: kc := 0: l := 0: lc := 0:
eqns(2,3);    yields
     eq2:,  D(s) = (r+rc)*s + (3*e-ec)*s + W0
  eq3:,  D(t) = (t+pc)*r + (tc+p)*s + (e-ec)*t + W1 + R01
```

9.7.5. suball – substitution utility for Newman-Penrose package

- The function suball attempts to make substitutions from the set of basic equations and their conjugates. The call "suball(expr);" will use all equations in the range 1..neqs ; a range may also be given as in "suball(expr,5,11);". In order to function properly, the user may need to rearrange the equations so that a single, atomic quantity appears on the left hand side. (See also: help(subs) .)
- Note: this will be slow if the range of equations is large; it helps to examine the equations first, and use limited ranges for substitution.

EXAMPLES :

```
with(np):
eqns():  k := 0 : kc := 0 :
eqns( 1, 1 );                          yields
  (eq1), D(r)  =  r^2 + s*sc + (e+ec)*r + R00
suball(D(r-rc),1,1): factor( " );    yields
  (r-rc)*(e+ec+r+rc)
```

9.8. The `numtheory` package for Number Theory

SYNTAX : numtheory[<function>] (args); or <function> (args);

- To use a numtheory function, type "with(numtheory, <function>);" to define <function>, or type "with(numtheory);" to define all numtheory functions.
- The functions available are:

bernoulli	binomial	cfrac	cyclotomic	divisors	easyfact
euler	factorset	fibonacci	ifactor	imagunit	isolve
isprime	isqrfree	ithprime	jacobi	lambda	legendre
mcombine	mlog	mobius	mroot	msqrt	nextprime
phi	pprimroot	prevprime	primroot	rootsunity	safeprime
tau	F	J	L	M	

- For help with a particular function do: help(numtheory, <function>);
- The package functions are always available without applying "with", using the long-form notation:
  numtheory[<function>] (<args>);
  This notation is necessary whenever there is a conflict between a package function name and another function used in the same session.

SEE ALSO : with

9.8.1. F – n'th Fermat number

CALLING SEQUENCE : F(n);
PARAMETERS : n – a non-negative integer

- The function F will compute the n'th Fermat number, which is 2^(2^n)+1.

EXAMPLES :

F(2); yields 17
F(3); yields 257

9.8.2. M – n'th Mersenne prime

CALLING SEQUENCE : M(n); or M([i]);
PARAMETERS : n, i – positive integers

- The function M will compute the n'th Mersenne prime. If the argument n is an integer then M(n) will return 2^n–1 if 2^n–1 is prime; otherwise it will return FAIL. If the argument is a list with one integer element [i] , then M([i]) will return the i'th Mersenne prime.

EXAMPLES :

```
M(3);     yields  7
M([3]);   yields  31
M(4);     yields  FAIL
```

9.8.3. cfrac – partial fraction convergents.

CALLING SEQUENCE : cfrac(r); or cfrac(r, maxit);
PARAMETERS : r – a rational or float number
maxit – (optional) integer (the number of convergents)

- This function prints the sequence of (maximum maxit) quotients and convergents of the number r.

SEE ALSO : help(convert,confrac)

9.8.4. cyclotomic – calculate cyclotomic polynomial

CALLING SEQUENCE : cyclotomic(n,t);
PARAMETERS : n – an integer
t – variable

- The function cyclotomic(n,t) will return the nth cyclotomic polynomial in t.

EXAMPLES :

```
cyclotomic(1,x);    yields  x-1
cyclotomic(20,z);   yields  z^8-z^6+z^4-z^2+1
```

9.8.5. divisors – positive divisors of an integer

CALLING SEQUENCE : divisors(n);
PARAMETERS : n – an integer

- The function divisors will compute the set of positive divisors of its integer argument n.

EXAMPLES :

```
divisors(-9);   yields  {1,3,9}
divisors(0);    yields  {}
divisors(1);    yields  {1}
divisors(10);   yields  {1,2,5,10}
```

9.8.6. factorset – prime factors of an integer

CALLING SEQUENCE : factorset(n);
PARAMETERS : n – an integer

- The function factorset will compute the set of prime factors of its integer argument n.

EXAMPLES :

```
factorset(10);   yields  {2,5}
factorset(96);   yields  {2,3}
```

SEE ALSO : ifactor

9.8.7. imagunit – square root of -1 mod n

CALLING SEQUENCE : imagunit(n);
PARAMETERS : n – an integer

- The function imagunit will find a square root of $-1 \pmod{n}$, if possible, otherwise it returns FAIL.

EXAMPLES :

```
imagunit(5);   yields  2
imagunit(7);   yields  FAIL
```

9.8.8. isqrfree – test if integer is square free

CALLING SEQUENCE : isqrfree(n);
PARAMETER : n – integer

- The function isqrfree(n) returns true if n is square free, and false otherwise. The integer n is square free if it is not divisible by a perfect square.

EXAMPLES :

```
isqrfree(20);   yields  false
isqrfree(21);   yields  true
```

9.8.9. jacobi – Jacobi symbol

CALLING SEQUENCE : jacobi(a,b); or J(a,b);
PARAMETERS : a – an integer relatively prime to b
b – a positive odd integer

- The function jacobi (synonym J) will compute the Jacobi symbol of a and b. If the factorization of b is b[1]*b[2]* ... *b[n], then J(a,b) = product(L(a,b[i]), i=1..n), where L(a,b) is the Legendre symbol of a and b.

EXAMPLES :

```
jacobi(6,11);   yields  -1
jacobi(7,9);    yields  1
```

9.8.10. lambda – Carmichael's lambda function

CALLING SEQUENCE : lambda(n);
PARAMETERS : n – a positive integer

- The size of the largest cyclic group generated by g^i (mod n) is given by lambda(n).
- Carmichael's theorem states that a^lambda(n) = 1 (mod n) if gcd(a,n)=1.

9.8.11. legendre – Legendre symbol

CALLING SEQUENCE : legendre(a,b); or legendre(a/b); or
L(a,b); or L(a/b);
PARAMETERS : a, b – integers

- The function legendre (synonym L) will compute the Legendre symbol of a and b, which is defined to be 1 if "a" is a quadratic residue (mod b) and –1 if "a" is a quadratic non-residue (mod b) . The number "a" is a quadratic residue of b if it has a square root (mod b); i.e., an integer c exists such that c^2 is congruent to a (mod b).

EXAMPLES :

```
legendre(7,9);   yields  1
legendre(3,5);   yields  -1
```

9.8.12. mcombine – Chinese remaindering

CALLING SEQUENCE : mcombine(a,ra,b,rb);
PARAMETERS : a, ra, b, rb – integers

- The function mcombine will compute an integer x such that x = ra (mod a) and x = rb (mod b), for a and b relatively prime, otherwise it returns FAIL.

EXAMPLES :

```
mcombine(6,5,9,7);    yields  FAIL
mcombine(7,4,11,5);   yields  60
```

9.8.13. mlog – discrete logarithm

CALLING SEQUENCE : mlog(x,a,n);
PARAMETERS : x, a, n – integers

- The function mlog will compute the discrete logarithm of x to the base a (mod n). It finds an integer y such that a^y = x (mod n) if possible, otherwise it returns FAIL .

EXAMPLES :

```
mlog(9,4,11);   yields  3
mlog(5,2,7);    yields  FAIL
```

9.8.14. mobius – Mobius function

CALLING SEQUENCE : mobius(n);
PARAMETERS : n – a positive integer

- The function mobius(n) gives the Mobius function of n (lattice of divisors).

9.8.15. mroot – modular root

CALLING SEQUENCE : mroot(x,r,p);
PARAMETERS : x, r, p – integers

- The function mroot will compute the rth root of x (mod p). It finds an integer y such that y^r = x (mod p) if possible, otherwise it returns FAIL .
- The order, r, of the root must be prime.

EXAMPLES :

```
mroot(5,11,13);   yields  8
mroot(2,3,6);     yields  2
```

9.8.16. msqrt – modular square root

CALLING SEQUENCE : msqrt(x,n);
PARAMETERS : x, n – integers

- The function msqrt will compute the square root of x (mod n). It finds an integer y such that y^2 = x (mod n) if possible, otherwise it returns FAIL .

EXAMPLES :

```
msqrt(3,11);   yields  5
msqrt(3,7);    yields  FAIL
```

9.8.17. phi – totient function

CALLING SEQUENCE : phi(n);
PARAMETERS : n – an integer

- The function phi will compute the totient function of n, which is the number of positive integers not exceeding n and relatively prime to n.

EXAMPLES :

```
phi(6);    yields  2
phi(15);   yields  8
```

9.8.18. pprimroot – compute a pseudo primitive root

CALLING SEQUENCE : pprimroot(g,n);
PARAMETERS : g – an integer
n – an integer greater than 2

- The function pprimroot(g,n) computes the next primitive root larger than g or, if n does not have primitive roots, computes a number which is not a root of order of any of the factors of phi(n).
- I.e. (in all cases), find an integer y, such that there is no x for which x^r = y (mod p) when r is a divisor of phi(p) greater than 1 and igcd(y,p)=1.

EXAMPLES :

```
pprimroot(1,41);   yields 6
pprimroot(2,8);    yields 3
```

SEE ALSO : help(numtheory,primroot)

9.8.19. primroot – compute a primitive root

CALLING SEQUENCE : primroot(g,p);
PARAMETERS : g – an integer
p – an integer greater than 2

- The function primroot will compute the first primitive root of p, that is greater than g, if possible, otherwise it returns FAIL . The integers that are relatively prime to p form a group of order phi(p) under multiplication (mod p). If this group is cyclic then a generator of the group is called a primitive root of p.

EXAMPLES :

```
primroot(1,41);   yields 6
primroot(2,8);    yields FAIL
```

SEE ALSO : help(numtheory,pprimroot)

9.8.20. rootsunity – roots of unity

CALLING SEQUENCE : rootsunity(r,p);
PARAMETERS : r – prime
p – integer

- This function will calculate all the rth roots of unity mod p and return the result as an expression sequence.
- The order, r, of the root must be prime.
- Note that there will always be at least the root 1.

EXAMPLES :

```
rootsunity(5,11);   yields  1,3,4,5,9
rootsunity(3,11);   yields  1
```

9.8.21. safeprime – compute a safe prime

CALLING SEQUENCE : safeprime(n);
PARAMETERS : n – an integer

- The function safeprime will compute the smallest safe prime that is greater than n. A safe prime is a number p such that p is prime and (p–1)/2 is prime.

EXAMPLES :

```
safeprime(8);    yields  11
safeprime(12);   yields  23
```

9.8.22. tau – Number of divisors

CALLING SEQUENCE : tau(n);
PARAMETERS : n – a positive integer

- The number of positive divisors of n is given by tau(n).

9.9. The `orthopoly` package for Orthogonal Polynomials

SYNTAX : orthopoly[<function>] (args); or <function> (args);

- To use an orthopoly function, type "with(orthopoly, <function>);" to define <function>, or type "with(orthopoly);" to define all orthopoly functions.
- The functions available are:

  G H L P T U
- G(n, a, x) generates the nth Gegenbauer polynomial.
- H(n, x) generates the nth Hermite polynomial.
- L(n, x) generates the nth Laguerre polynomial.
- L(n, a, x) generates the nth generalized Laguerre polynomial.
- P(n, x) generates the nth Legendre polynomial.
- P(n, a, b, x) generates the nth Jacobi polynomial.
- T(n, x) generates the nth Chebyshev polynomial of the first kind.
- U(n, x) generates the nth Chebyshev polynomial of the second kind.
- For a more detailed description of a particular function do:

  help(orthopoly, <function>);
- The package functions are always available without applying "with", using the long-form notation:

  orthopoly[<function>] (<args>);

  This notation is necessary whenever there is a conflict between a package function name and another function used in the same session.

SEE ALSO : with

9.9.1. G – n'th Gegenbauer (ultraspherical) polynomial

CALLING SEQUENCE : G(n,a,x);

PARAMETERS :
- n – a non-negative integer
- a – any non-rational algebraic expression or a rational number a > –1/2
- x – any algebraic expression

- The function G will compute the n'th ultraspherical (Gegenbauer) polynomial, with parameter a, evaluated at x.
- These polynomials are orthogonal on the interval [–1,1], with respect to the weight function w(x) = (1–x^2)^(a–1/2); that is, int(w(x)*G(n,a,x)*G(m,a,x),x=–1..1) = 0 if m is not equal to n.

- They satisfy the following recurrence relation, for "a" non-zero:
  G(0,a,x) = 1, G(1,a,x) = 2*a*x,
  G(n,a,x) = 2*(n+a–1)/n*x*G(n–1,a,x)
  – (n+2*a–2)/n*G(n–2,a,x), for n>1.
  When a=0, G(0,0,x) = 1, and G(n,0,x) = 2/n*T(n,x), for n>0.

EXAMPLES :

```
G(2,5,x);   yields  60*x^2 – 5
G(3,2,x);   yields  32*x^3 – 12*x
```

9.9.2. H – n'th Hermite polynomial

CALLING SEQUENCE : H(n,x);
PARAMETERS : n – a non-negative integer
x – any algebraic expression

- The function H will compute the n'th Hermite polynomial, evaluated at x.
- These polynomials are orthogonal on the interval (–infinity, infinity), with respect to the weight function w(x) = exp(–x^2); that is, int(w(x)*H(n,x)*H(m,x),x=–infinity..infinity) = 0 if m is not equal to n.
- They satisfy the following recurrence relation: H(0,x) = 1, H(1,x) = 2*x, H(n,x) = 2*x*H(n–1,x) – 2*(n–1)*H(n–2,x), for n>1.

EXAMPLES :

```
H(2,x);   yields  4*x^2 – 2
H(3,x);   yields  8*x^3 – 12*x
```

9.9.3. L – n'th Laguerre polynomial

CALLING SEQUENCE : L(n,a,x); or L(n,x);
PARAMETERS : n – a non-negative integer
a – any non-rational algebraic expression
or a rational number a > –1
x – any algebraic expression

- The function L will compute the n'th generalized Laguerre polynomial, with parameter a, evaluated at x. In the case of only two arguments, L(n,x); will find the n'th Laguerre polynomial, which is equal to L(n,0,x).

- These polynomials are orthogonal on the interval [0,infinity), with respect to the weight function w(x) = exp(–x)*x^a; that is, int(w(x)*L(n,a,x)*L(m,a,x),x=0..infinity) = 0 if m is not equal to n.
- They satisfy the following recurrence relation:
  L(0,a,x) = 1,
  L(1,a,x) = –x+1+a,
  L(n,a,x) = (2*n+a–1–x)/n*L(n–1,a,x) – (n+a–1)/n*L(n–2,a,x)
  for n>1 .

EXAMPLES :

```
L(3,x);     yields  –1/6*x^3 + 3/2*x^2 – 3*x + 1
L(2,1,x);   yields  (1/2)*x^2 – 3*x + 3
```

9.9.4. P – Legendre and Jacobi polynomials

CALLING SEQUENCE : P(n,a,b,x); or P(n,x);

PARAMETERS :
- n – a non-negative integer
- a, b – any non-rational algebraic expressions or rational numbers greater than –1
- x – any algebraic expression

- The function P will compute the n'th Jacobi polynomial, with parameters a and b, evaluated at x. In the case of only two arguments, P(n,x); will find the n'th Legendre (spherical) polynomial, which is equal to P(n,0,0,x).
- These polynomials are orthogonal on the interval [–1,1], with respect to the weight function w(x) = (1–x)^a*(1+x)^b; that is, int(w(x)*P(n,a,b,x)*P(m,a,b,x),x=–1..1) = 0 if m is not equal to n.
- They satisfy the following recurrence relation:
  P(0,a,b,x) = 1,
  P(1,a,b,x) = (a/2–b/2) + (1+a/2+b/2)*x,
  P(n,a,b,x) = (2*n+a+b–1–x) * (a^2–b^2+(2*n+a+b–2)
  * (2*n+a+b)*x) / (2*n*(n+a+b)*(2*n+a+b–2)) * P(n–1,a,b,x)
  – 2*(n+a–1)*(n+b–1)*(2*n+a+b)
  / (2*n*(n+a+b)*(2*n+a+b–2)) * P(n–2,a,b)
  for n>1 .

EXAMPLES :

```
P(3,x);          yields  5/2*x^3 – 3/2*x
P(2,1,3/4,x);    yields  437/128*x^2 + 19/64*x – 91/128
```

9.9.5. T – n'th Chebyshev polynomial (of the first kind)

CALLING SEQUENCE : T(n,x);
PARAMETERS : n – a non-negative integer
x – any algebraic expression

- The function T will compute the n'th Chebyshev polynomial of the first kind, evaluated at x.
- These polynomials are orthogonal on the interval [–1, 1], with respect to the weight function w(x) = (1–x^2)^(–1/2); that is, int(w(x)*T(n,x)*T(m,x),x=–1..1) = 0 if m is not equal to n.
- They satisfy the following recurrence relation: T(0,x) = 1, T(1,x) = x, T(n,x) = 2*x*T(n–1,x) – T(n–2,x), for n>1.

EXAMPLES :

```
T(2,x);   yields  2*x^2 – 1
T(3,x);   yields  4*x^3 – 3*x
```

9.9.6. U – n'th Chebyshev polynomial (of the second kind)

CALLING SEQUENCE : U(n,x);
PARAMETERS : n – a non-negative integer
x – any algebraic expression

- The function U will compute the n'th Chebyshev polynomial of the second kind, evaluated at x.
- These polynomials are orthogonal on the interval [–1, 1], with respect to the weight function w(x) = (1–x^2)^(1/2); that is, int(w(x)*U(n,x)*U(m,x),x=–1..1) = 0 if m is not equal to n.
- They satisfy the following recurrence relation: U(0,x) = 1, U(1,x) = 2*x, U(n,x) = 2*x*U(n–1,x) – U(n–2,x), for n>1.

EXAMPLES :

```
U(2,x);   yields  4*x^2 – 1
U(3,x);   yields  8*x^3 – 4*x
```

9.10. The `powseries` package for Formal Power Series

SYNTAX : powseries[<function>] (args); or <function> (args);

- To use a powseries function, type "with(powseries, <function>);" to define <function>, or type "with(powseries);" to define all powseries functions.
- The functions available are:

add	compose	evalpow	inverse	multconst	multiply
negative	powcreate	powdiff	powexp	powint	powlog
powpoly	powsolve	quotient	reversion	subtract	tpsform

- For help with a particular function do: help(powseries, <function>);
- E.g., to multiply two power series A and B, type:
  with(powseries,multiply); multiply(A,B);
- The package functions are always available without applying "with", using the long-form notation:
  powseries[<function>] (<args>);
  This notation is necessary whenever there is a conflict between a package function name and another function used in the same session.
- For help with the general representation of formal power series, do: help (powseries, powseries);

SEE ALSO : with

9.10.1. add – addition of formal power series

CALLING SEQUENCE : add(A,B,..,Z)
PARAMETERS : A, B,..,Z – an arbitrary number of formal power series

- add(A,B,..,Z) returns the formal power series which is sum of A,B,..,Z

EXAMPLES :

```
with(powseries):
powcreate(t(n)=t(n–1)/n,t(0)=1):
powcreate(v(n)=v(n–1)/2,v(0)=1):
s := add(t, v):
tpsform(s, x, 7);   yields
2+3/2*x+3/4*x^2+7/24*x^3+5/48*x^4+19/480*x^5+49/2880*x^6+O(x^7)
```

9.10.2. compose – composition of formal power series

CALLING SEQUENCE : compose(a, b)
PARAMETERS : a, b – formal power series

- compose(a, b) returns the formal power series which is the composition of a and b (a(b)); i.e., a(0) + a(1)*b + a(2)*b^2 + a(3)*b^3 + ...
- b(0) must be 0, otherwise the composition is not well defined, and an error message is returned.

EXAMPLES :

```
with(powseries):
powcreate(t(n)=t(n-1)/n,t(0)=1):
powcreate(v(n)=v(n-1)/2,v(0)=0,v(1)=1):
s := compose (t, v):
tpsform(s, x);   yields
    1 + x + x^2 + 11/12*x^3 + 19/24*x^4 + 157/240*x^5 + O(x^6)
```

9.10.3. evalpow – general evaluator for power series expressions

CALLING SEQUENCE : evalpow(expr);
PARAMETERS : expr – any arithmetic expression involving powerseries and constants.

- evalpow evaluates any arithmetic expression involving powerseries and constants. It returns an unnamed powerseries.
- the operators +,–,*,/,^ can be used as well as composition of functions i.e. f(g).
- the other functions that can be used in evalpow are: pexp,pinv,plog,pneg,prev (reversion),pdiff (first derivative), pint (first integral),pquo (quotient),psub (subtract).
- for help on any particular function type help(powseries,<function>);

EXAMPLES :

```
with(powseries):
powcreate(f(n)=f(n-1)/n,f(0)=1):
powcreate(g(n)=g(n-1)/2,g(0)=0,g(1)=1):
powcreate(h(n)=h(n-1)/5,h(0)=1):
k:=evalpow(f^3+g-pquo(h,f)):
tpsform(k,x,5);   yields
  24/5*x+233/50*x^2+7273/1500*x^3+52171/15000*x^4+O(x^5)
powcreate(t(n)=1/(n-5), t(5)=0):
powcreate(u(n)=u(n-1)+u(n-2),u(0)=0,u(1)=1):
v := evalpow( (t+3*u)/t ):
```

tpsform(v,x); yields
1−15*x+15/4*x^2−155/16*x^3−105/64*x^4+6775/768*x^5+O(x^6)
SEE ALSO : help(powseries,powcreate)

9.10.4. inverse – multiplicative inverse of a formal power series

CALLING SEQUENCE : inverse(powser)
PARAMETERS : powser – formal power series

- inverse(powser) returns a formal power series which is the inverse of powser with respect to multiply.
- I.e., multiply(powser, inverse(powser)) will return the formal power series equivalent to 1.

EXAMPLES :
with(powseries):
powcreate(t(n)=t(n−1)/n,t(0)=1):
s := inverse(t):
tpsform(s, x, 7); yields
1 − x + 1/2*x^2 − 1/6*x^3 + 1/24*x^4 − 1/120*x^5 + 1/720*x^6 + O(x^7)

9.10.5. multconst – multiplication of a power series by a constant

CALLING SEQUENCE : multconst(powser, const)
PARAMETERS : powser – formal power series
const – scalar constant

- multconst(powser, const) returns the formal power series where each coefficient is const times the coefficient in powser.

EXAMPLES :
with(powseries):
powcreate(t(n)=t(n−1)/n,t(0)=1):
s := multconst(t, −42):
tpsform(s, x); yields
− 42 − 42*x − 21*x^2 − 7*x^3 − 7/4*x^4 − 7/20*x^5 + O(x^6)

9.10.6. multiply – multiplication of formal power series

CALLING SEQUENCE : multiply(a, b)
PARAMETERS : a, b – formal power series

- multiply(a, b) returns the formal power series which is the product of a and b.

EXAMPLES :
```
with(powseries):
powcreate(t(n)=t(n-1)/n,t(0)=1):
powcreate(v(n)=v(n-1)/2,v(0)=1):
s := multiply( t, v ):
tpsform(s, x);   yields
  1 + 3/2*x + 5/4*x^2 + 19/24*x^3 + 7/16*x^4 + 109/480*x^5 + O(x^6)
```

9.10.7. negative – negation of a formal power series

CALLING SEQUENCE : negative(powser)
PARAMETERS : powser – formal power series

- negative(powser) returns a formal power series where each coefficient is the negative of the coefficient in powser.

EXAMPLES :
```
with(powseries):
powcreate(t(n)=t(n-1)/n,t(0)=1):
s := negative(t):
tpsform(s, x);   yields
  -1 - x - 1/2*x^2 - 1/6*x^3 - 1/24*x^4 - 1/120*x^5 + O(x^6)
```

9.10.8. powcreate – create formal power series

CALLING SEQUENCE : powcreate(recursive eqn, <initial conditions>)
PARAMETERS : eqn – recursive equation specifying coefficients of formal series, where coefficient of x^i is f(i)
init – the initial coefficients (if necessary)

- powcreate(t(n)=expr, t(0)=a1, ... t(m)=am) creates a formal power series, t, whose n-th term is expr, and whose initial terms are t(0) = a1 ,..., t(m) = am .
- it is possible to include arbitrary conditions in eqns, to override the expr in specific cases.

EXAMPLES :
```
with(powseries):
powcreate(t(n)=1/n!,t(0)=1):
tpsform(t, x, 7);   yields
  1+x+1/2*x^2+1/6*x^3+1/24*x^4+1/120*x^5+1/720*x^6+O(x^7)
powcreate(v(n)=(v(n-1)+v(n-2))/4,v(0)=4,v(1)=2):
tpsform(v, x);   yields
  4+2*x+3/2*x^2+7/8*x^3+19/32*x^4+47/128*x^5+O(x^6)
```

9.10.9. powdiff – differentiation of a formal power series

CALLING SEQUENCE : powdiff(powser)
PARAMETERS : powser – formal power series

- powdiff(powser) returns the formal power series which is the derivative of powser with respect to the variable of powser.

EXAMPLES :

```
with(powseries):
t := powpoly(x+1, x):
s := powlog(t):
tpsform(s, x, 6);    yields
   x - 1/2*x^2 + 1/3*x^3 - 1/4*x^4 + 1/5*x^5 + O(x^6)
r := powdiff(s):
tpsform(r, x, 10);    yields
   1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + O(x^10)
```

9.10.10. powexp – exponentiation of a formal power series

CALLING SEQUENCE : powexp(powser)
PARAMETERS : powser – formal power series

- powexp(powser) returns the formal power series which is exp(powser).

EXAMPLES :

```
with(powseries):
t := powpoly(x, x):
s := powexp(t):
tpsform(s, x, 7);    yields
  1+x+1/2*x^2+1/6*x^3+1/24*x^4+1/120*x^5+1/720*x^6+O(x^7)
```

9.10.11. powint – integration of a formal power series

CALLING SEQUENCE : powint(powser)
PARAMETERS : powser – formal power series

- powint(powser) returns the formal power series which is the integral of powser with respect to the variable of the power series.

EXAMPLES :

```
with(powseries):
powcreate(t(n)=1/n,t(0)=1):
tpsform(t, x, 5);    yields
   1 + x + 1/2*x^2 + 1/3*x^3 + 1/4*x^4 + O(x^5)
r := powint(t):
tpsform(r, x, 7);    yields
 x + 1/2*x^2 + 1/6*x^3 + 1/12*x^4 + 1/20*x^5 + 1/30*x^6 + O(x^7)
```

9.10.12. powlog – logarithm of a formal power series

CALLING SEQUENCE : powlog(powser)
PARAMETERS : powser – formal power series

- powlog(powser) returns the formal power series which is the natural logarithm of powser.
- powser must have a non-zero first term (powser(0) <> 0), otherwise it cannot be represented as a power series.

EXAMPLES :

```
with(powseries):
t := powpoly(1 + x, x):
s := powlog(t):
tpsform(s, x, 7);    yields
   x – 1/2*x^2 + 1/3*x^3 – 1/4*x^4 + 1/5*x^5 – 1/6*x^6 + O(x^7)
```

9.10.13. powpoly – create a formal power series from a polynomial

CALLING SEQUENCE : powpoly(poly, var)
PARAMETERS : poly, var – poly is a polynomial in the variable var

- powpoly(poly, var) returns a formal power series equivalent to poly; i.e., the i-th coefficient is coeff(poly, var, i) for all i.

EXAMPLES :

```
with(powseries):
t := powpoly(2*x^5+4*x^4–3*x^2+5, x):
tpsform(t, s, 5);    yields
      5 – 3*s^2 + 4*s^4 + 2*s^5
tpsform(t, s, 4);    yields
        5 – 3*s^2 + O(s^4)
```

9.10.14. powsolve – solve a linear D.E. as a formal power series

CALLING SEQUENCE : powsolve(sys)
PARAMETERS : sys – a set containing a linear differential equation and possibly initial conditions,

- "powsolve" solves a linear differential equation, possibly with initial conditions.
- initial conditions must be at zero, derivatives are denoted by appending p's to the function name, e.g. 2nd derivative of y at 0 is ypp(0)
- The solution returned is a formal power series which represents the infinite series solution.

EXAMPLES :

```
with(powseries):
a:=powsolve({diff(y(x),x)=y(x),y(0)=1}):
tpsform(a, x);    yields
   1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
# second system
v:=powsolve({diff(y(x),x$4)=y(x),y(0)=3/2,yp(0)=-1/2,
 ypp(0)=-3/2,yppp(0)=1/2}):
tpsform(v, x);    yields
   3/2 - 1/2*x - 3/4*x^2 + 1/12*x^3 + 1/16*x^4 - 1/240*x^5 + O(x^6)
```

9.10.15. quotient – quotient of two formal power series

CALLING SEQUENCE : quotient(a, b)
PARAMETERS : a, b – formal power series

- quotient(a, b) returns the formal power series which is a/b .
- quotient(a, b) yields the same result as, but is faster than, multiply(a, inverse(b)) .

EXAMPLES :

```
with(powseries):
powcreate(t(n)=t(n-1)/n,t(0)=1):
powcreate(v(n)=v(n-1)/5,v(0)=1):
s := quotient(t, v):
tpsform(s, x, 4);    yields
      1 + 4/5*x + 3/10*x^2 + 1/15*x^3 + O(x^4)
```

9.10.16. representation – representation of formal power series

SYNTAX : powseries[<function>](args); or <function>(args);

- Formal power series are procedures which return the coefficients of the series; i.e., name(i) is the coefficient of x^i in the series named "name".
- The values of the coefficients are saved using a remember table, so all computed coefficients can be seen via: op(4, op(name)) .
- The actual procedure is identical for all power series; the only differences are the general term and the values each remembers.
- The general term of the power series can be obtained via: name(_k) .
- Each intermediate power series created in a calculation should be named.

EXAMPLES :

```
with(powseries):
powcreate(e(n)=1/n!,e(0)=1):
powcreate(f(n)=f(n-1)/(n^2*f(n-2)),f(0)=1,f(1)=5,f(2)=2):
tpsform(e, x, 6);          yields
   1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
tpsform(f, x, 6);          yields
  1 + 5*x + 2*x^2 + 2/45*x^3 + 1/720*x^4 + 1/800*x^5 + O(x^6)
# Compute the series f^e .
logf := powlog(f):
elogf := multiply(e, logf):
result := powexp(elogf):
tpsform(result, x, 6);    yields
1 + 5*x + 7*x^2 + 767/45*x^3 + 2351/240*x^4 + 39231/800*x^5 + O(x^6)
```

9.10.17. reversion – reversion of formal power series

CALLING SEQUENCE : reversion(a, b); or reversion(a);

PARAMETERS : a – formal power series
b – (optional) formal power series

- reversion(a, b) returns the formal power series which is the reversion of a with respect to b. If b is not specified then the series is assumed to be 0 + 1*x + 0*x^2 + ...
- Reversion is the inverse of composition, so that composition of the result into a will give b. I.e., solve a(r) = b for the power series r.

- a(0) must be 0, a(1) must be 1, and b(0) must be 0; otherwise the reversion is not well defined and an error message is returned.

EXAMPLES :

```
with(powseries):
powcreate(t(n)=t(n-1)/n,t(0)=0,t(1)=1):
powcreate(v(n)=v(n-1)/2,v(0)=0,v(1)=1):
s := reversion(t,v):
tpsform(s,x,11);    yields
   x + 1/12*x^3 + 1/80*x^5 + 1/448*x^7 + 1/2304*x^9 + O(x^11)
ts := compose(t,s):
tpsform(ts,x);      yields
   x + 1/2*x^2 + 1/4*x^3 + 1/8*x^4 + 1/16*x^5 + O(x^6)
tpsform(v,x);       yields
   x + 1/2*x^2 + 1/4*x^3 + 1/8*x^4 + 1/16*x^5 + O(x^6)
```

9.10.18. subtract – subtraction of two formal power series

CALLING SEQUENCE : subtract(a, b)
PARAMETERS : a, b – formal power series

- subtract(a, b) returns the formal power series in which each coefficient is the coefficient of "a" minus the coefficient of "b".
- subtract(a, b) yields the same result as, but is faster than, add(a, negative(b)) .

EXAMPLES :

```
with(powseries):
powcreate(t(n)=t(n-1)/n,t(0)=1):
powcreate(v(n)=v(n-1)/2,v(0)=1):
s := subtract(v, t):
tpsform(s, x, 6);    yields
  - 1/2*x - 1/4*x^2 - 1/24*x^3 + 1/48*x^4 + 11/480*x^5 + O(x^6)
```

9.10.19. tpsform – truncated power series form of a formal series

CALLING SEQUENCE : tpsform(powser, var, order); or
tpsform(powser, var);
PARAMETERS : powser – name of a formal power series
var – variable to be used in the truncated series
order – (optional) result will be accurate to order
(default is the global variable Order)

- tpsform(powser, var, order) creates a truncated power series in var which is equal to powser up to x^order.
- In some cases (e.g., formal power series from powpoly), tpsform can determine when the series is exact and will drop the O(x^order) term.

EXAMPLES :

```
with(powseries):
powcreate(t(n)=t(n-1)/n,t(0)=1):
t(_k);                yields  t(_k-1)/_k
tpsform(t, x, 6);     yields
  1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
```

9.11. The `simplex` package for Linear Optimization

SYNTAX : simplex[<function>] (args); or <function> (args);

- The simplex package is a collection of routines designed for linear optimization. The implementation of the simplex algorithm used is based on the initial chapters of "Linear Programming" by Chvatal. In addition to the routines feasible(), maximize() and minimize(), the package provides routines to assist the user in carrying out the steps of the algorithm one at a time, setting up problems, etc.
- The functions available are:

basis convexhull cterm dual feasible maximize
minimize pivot pivoteqn pivotvar ratio setup
standardize variables

- Type help(simplex,f); for help on a particular function f.
- To maximize a linear function f subject to the set of linear constraints c, type:

with(simplex); maximize(f,c);

- The "with" function must be invoked to use the short-form notation. The long form of the notation can always be used to resolve name conflicts between a package function and another function used in the same session.

EXAMPLES :

```
with(simplex);
cnsts := {3*x+4*y-3*z <= 23, 5*x-4*y-3*z <= 10,
                                7*x+4*y+11*z <= 30}:
obj := -x + y + 2*z:
maximize(obj,cnsts union {x>=0,y>=0,z>=0});
                                {x=0, y=49/8, z=1/2}
maximize(obj,cnsts,NONNEGATIVE);
                                {x=0, y=49/8, z=1/2}
maximize(obj,cnsts);
                                NULL  (unbounded)
```

9.11.1. basis

CALLING SEQUENCE : basis(C) (or simplex[basis](C))
PARAMETERS : C – a set of linear equations in the special form produced by simplex[setup]()

- basis(C) computes a list of variables, one per equation, which corresponds to the basis used by the simplex algorithm. setup() ensures that each of these variables occurs in exactly one equation, and only on the left hand side.

EXAMPLES :
basis([x = 3*y + z , w = 2*y – z]); ==> [x,w]

9.11.2. convexhull – Find convex hull enclosing the given points

CALLING SEQUENCE : simplex[convexhull](ps)
PARAMETERS :
ps – a list or set of points. Every point is a list of an x and y coordinate (both have to be numeric).

- This function may be loaded via: with(simplex,convexhull); or else it may be invoked directly using the notation simplex[convexhull](ps).
- The result is returned as a list of points in counter-clockwise order.
- Uses an n*log(n) algorithm computing tangents of pairs of points.

EXAMPLES :
with(simplex,convexhull);
convexhull({ [0,0], [1,1], [2,0], [1,0], [1,1/2] });
returns: [[0,0], [2,0], [1,1]]

9.11.3. cterm

CALLING SEQUENCE : cterm(C); (or simplex[cterm](C);)
PARAMETERS : C – a set or list of linear equations or inequalities

- cterm(C) computes a list of constants, one per equation or inequality. These are computed as if the constants were collected on the right hand side of the equation or inequality.

EXAMPLES :
cterm([3*x + y <= 5 , 4*y – z – 3 = 3]); ==> [5,6]

9.11.4. dual

CALLING SEQUENCE : dual(f,C,y); (or simplex[dual](f,C,y))
PARAMETERS : f – a linear objective function to be maximized
C – a set of linear inequalities in the special form produced by convert(C,stdle);
y – a name to be used in naming the dual variables

- dual(f,C,y) computes the dual of a linear program which is in standard inequality form. The name y is used to construct the names y1, y2, ... for the dual variables. The resulting dual is returned as an expression sequence: objective, constraints .

EXAMPLES :
```
dual( x+y, {3*x+4*y <= 4, 4*x+3*y <= 5}, z );    ==>
                    4*z1+5*z2, {1 <= 3*z1+4*z2, 1 <= 4*z1+3*z2}
```

9.11.5. equality

CALLING SEQUENCE : convert(s,equality);
PARAMETERS : s – a list or set of equalities (and /or) inequalities

- All equations and inequalities are converted to type `=`. Note that the feasible region described by the original set of relations is usually not preserved by this operation.

EXAMPLES :
```
convert( {3*x+4*y <= 4, 4*x+3*y <= 5}, equality );  ==>
          { 3*x+4*y = 4, 4*x+3*y = 5 }
```

9.11.6. feasible

CALLING SEQUENCE : simplex[feasible](C); simplex[feasible](C,vartype);
simplex[feasible](C,vartype,'NewC','Transform')
PARAMETERS : C – a set of linear constraints
Optional parameters:
vartype – parameter to indicate non-negativity constraints (e.g. NONNEGATIVE, UNRESTRICTED)
NewC – a name for returning a set of constraints in a form used to recognize feasibility
Transform – a name for returning any variable transformations required in setting up NewC

- feasible(C); returns true or false according as a feasible solution to the linear system exists or not. Non-negativity constraints on all the variables can be indicated by use of a second argument: NONNEGATIVE or by explicitly listing the constraints. The final two arguments are used to return as sets, the final system found by feasible, and any variable transformations which occurred. The new system may have global "artificial" and "slack" variables present (e.g. _AR, or _SL1, ...).

EXAMPLES :
```
feasible({4*x+3*y <= 5, 3*x+4*y = 4}, NONNEGATIVE);  ==> true
feasible({4*x-3*y <= 5, 3*x-4*y = 4, x>=0, y>=0 });  ==> false
```

9.11.7. maximize

CALLING SEQUENCE : maximize(f , C); or
maximize(f , C , NONNEGATIVE);
maximize(f , C , UNRESTRICTED , 'NewC' ,'transform');

PARAMETERS : f – a linear objective function
C – a set or list of linear constraints
Type – to specify additional constraints on variables
NewC – name to return final description of system
transform – name to return set of variable transformations used to get NewC

- maximize(f , C) returns either a set of equations describing the optimal solution to the specified linear program, or NULL. The function returns the empty set in the case where no feasible solution to C exists. The solution is NULL in the case where the solution is unbounded.
- Use subs() to substitute the solution into the original constraints, or to obtain the objective function value at the solution.
- A third parameter may be used to specify that all variables are constrained to be NONNEGATIVE. Such constraints may also be listed explicitly.
- A fourth and a fifth parameter may be included to specify names for returning the optimal description, and any variable transformations used to set up the problem.

EXAMPLES :

```
maximize( x+y, {4*x+3*y <= 5, 3*x+4*y <= 4 } );
                                {x = 8/7, y = 1/7}
maximize( x–y, {3*x+4*y <= 4, 4*x+3*y <= –3} );
                                 NULL (unbounded)
maximize( x–y, {3*x+4*y <= 4, 4*x+3*y <= –3}, NONNEGATIVE );
                                {}  (infeasible)
```

9.11.8. minimize

CALLING SEQUENCE : minimize(f , C); or
minimize(f , C , NONNEGATIVE);
minimize(f , C , UNRESTRICTED , 'NewC' ,'transform');

PARAMETERS : f – a linear objective function
C – a set or list of linear constraints
Type – to specify additional constraints on variables
NewC – name to return final description of system

transform – name to return set of variable transformations used to get NewC

- minimize(f , C) returns either a set of equations describing the optimal solution to the specified linear program, or NULL. The function returns the empty set in the case where no feasible solution to C exists. The solution is NULL in the case where the solution is unbounded. If printlevel is set to be greater than 1, a warning is printed for the unbounded case.
- See help(simplex , maximize) for details on optional parameters.
- Use subs() to substitute the solution into the original constraints, or to obtain the objective function value at the solution.

EXAMPLES :
```
minimize( x+y, {4*x+3*y <= 5, 3*x+4*y <= 4}, NONNEGATIVE );
                                    { x=0 , y=0 }
```

9.11.9. nonneg

CALLING SEQUENCE : type(expr , nonneg)
PARAMETERS : expr – any expression

- This is a boolean function to test if the expression is a constraint of the form: a >= 0 .

EXAMPLES :
```
type( -a <= 0 , nonneg );   ==>  true
type(  0 <= a , nonneg );   ==>  true
type(  a = 0 , nonneg );    ==>  false
type(  0 <= a+b , nonneg ); ==>  false
```

9.11.10. pivot

CALLING SEQUENCE : pivot(C , x , eqn);
PARAMETERS : C – a set or list of equations in the special form obtained by simplex[setup]()
x – the variable to be added to the basis
eqn – an equation or list of equations (contained in the set C) whose variable on the left hand side is to leave the basis

- pivot(C, x , eqn) constructs a new set of equations in a form compatible with simplex[setup](), by solving the specified equation eqn, for x, and substituting the result into C. This is equivalent to the standard simplex pivot on an array of coefficients.

- To reexpress the objective function f in terms of this new description, use subs( ) to substitute the result of pivot() into f.
- If eqn is a list of equations, then eqn[1] is used. This is to facilitate easy user interaction with the function pivoteqn() used to choose an equation to use for the next pivot.

EXAMPLES :
```
pivot( {_SL1 = 5–4*x–3*y, _SL2 = 4–3*x–4*y}, x,
                        [_SL1 = 5–4*x–3*y] );
     {x = 5/4–3/4*y–1/4*_SL1, _SL2 = 1/4–7/4*y+3/4*_SL1}
```

9.11.11. pivoteqn

CALLING SEQUENCE : pivoteqn (C , var);
PARAMETERS : C – a set of linear equations in the special form produced by simplex[setup]()
var – the variable chosen to carry out a pivot operation

- pivoteqn(C, var) returns a sublist of equations from C. Each equation eq returned achieves the minimum non-negative ratio of
(– cterm(eq)) / coeff(eq, var, 1) .
In the special case where all ratios are negative, the function returns FAIL.

EXAMPLES :
```
pivoteqn( {_SL1 = 5–4*x–3*y, _SL2 = 4–3*x–4*y}, x );  ==>
          [_SL1 = 5–4*x–3*y]
```

9.11.12. pivotvar

CALLING SEQUENCE : pivotvar(f , List);
pivotvar(f);
PARAMETERS : f – an objective function expressed in terms of non-basic variables relative to some description
List – a list of the variables which occur in the problem

- pivotvar(f, List) returns either a variable which has a positive coefficient or FAIL. The optional List (of variable names) is used to determine the order in which the variables should be examined. The first success is used. If no List is provided, then pivotvar() chooses its own order.

EXAMPLES :
```
pivotvar( x1 + 3*x3 – x4 );                    ==>  x1   ( possibly  x3 )
pivotvar( x1 + 3*x3 – x4 , [x4,x3,x1] );  ==>  x3
```

9.11.13. ratio

CALLING SEQUENCE : ratio(C , x);
ratios(C , x);
PARAMETERS : C – a set of linear equations in the special form produced by simplex[setup]()
x – a variable to be used in calculating ratios

- ratio(C, x) returns a list of ratios which can be used to determine which equation in C is most restrictive with respect to the next simplex pivot. For each equation, the ratio computed, if possible, is
  – cterm(eq) / coeff(eq, x, 1) .
  All negative ratios are reported as infinity, as are those for which the denominator is 0.

EXAMPLES :
ratio([_SL1 = 5–4*x–3*y, _SL2 = 4–3*x–4*y], x); ==> [5/4, 4/3]

9.11.14. setup

CALLING SEQUENCE : setup(C);
setup(C , NONNEGATIVE);
setup(C , NONNEGATIVE , 't');
PARAMETERS : C – a set of linear equations in the special form produced by simplex[setup]()
NONNEGATIVE – to indicate that all variables are assumed to be restricted to be non-negative
't' – a name used to return the transformations used for any variables deemed to be unrestricted

- setup(C) constructs a set of equations in a form with isolated variables on the left hand side. Those variables form a basis for the corresponding linear system, and do not occur on the right hand side of any equation. Slack variables of the form _SL.i are introduced to deal with inequalities. Unrestricted variables are transformed to be the difference of two variables.
- The resulting system is equivalent to the original system C, in that solutions to the new system correspond can be transformed into solutions of the original system. The resulting system need not correspond to a feasible basic solution.

EXAMPLES :
setup({3*x+4*y <= 4, 4*x+3*y = 5}); ==>
{y = –4/3*x+5/3, _SL1 = –8/3+7/3*x}

9.11.15. std

CALLING SEQUENCE : convert(C , std);
PARAMETERS : C – a set or list of linear equations and inequalities

- convert(C, std) returns a set (or list) of constraints obtained by moving all constants to the right hand side of the equation (inequality) for each constraint in C.
- No attempt is made to represent equations by inequalities.
- Note that inequalities of the form ">=" are automatically transformed by Maple into the form "<=".

EXAMPLES :
```
convert( {4 <= 3*x+4*y, 5 <= 4*x+3*y}, std );  ==>
        {-3*x-4*y <= -4, -4*x-3*y <= -5}
```

9.11.16. standardize

CALLING SEQUENCE : standardize(C);
PARAMETERS : C – a set or list of linear constraints

- standardize(C) returns a set or list of inequalities of the form `<=`, equivalent to the original set of constraints, with all constants on the right hand side. See also: help(simplex,stdle).
- It differs from convert(C,std) in that equations are transformed into pairs of inequalities.
- It is equivalent to convert(C,stdle).

EXAMPLES :
```
standardize( {3*x+4*y <= 4, 4*x+3*y = 5} );  ==>
        {-4*x-3*y <= -5, 3*x+4*y <= 4, 4*x+3*y <= 5}
```

9.11.17. stdle

CALLING SEQUENCE : convert(C , stdle);
PARAMETERS : C – a set or list of linear constraints

- convert(C, stdle) returns a set or list of inequalities of the form `<=`, equivalent to the original set of constraints, with all constants on the right hand side. See also: help(simplex,standardize).

EXAMPLES :
```
convert( {3*x+4*y <= 4, 4*x+3*y = 5} , stdle );  ==>
        {-4*x-3*y <= -5, 3*x+4*y <= 4, 4*x+3*y <= 5}
```

9.11.18. variables

CALLING SEQUENCE : variables(C);
PARAMETERS : C – a set of linear constraints

- variables(C) returns the set of variables which occur in C.

EXAMPLES :
variables({3*x+4*y <= 4, 4*x+3*y = 5}); ==> {x, y}

9.12. The `stats` package for Statistics

SYNTAX : stats[<function>] (args); or <function> (args);

- To use a stats function, type "with(stats, <function>);" to define <function>, or type "with(stats);" to define all stats functions.
- The stats package allows simple statistical manipulation of data. This includes: averaging, standard deviation, correlation coefficients, variance, regression analysis, etc.
- The following statistical procedures have been implemented:

ChiSquare	Exponential	Fdist	Ftest	N
Rsquared	StudentsT	Uniform	average	correlation
covariance	linregress	median	mode	multregress
projection	regression	sdev	serr	statplot
variance				

- The following are help files concerned with the statistical matrix and user defined distributions.

addrecord	data	distribution	evalstat	getkey
putkey	removekey			

- For help with a particular function do: help(stats, <function>);
- The package functions are available without applying "with", using the long-form notation:

  stats[<function>] (<args>);

  This notation is necessary whenever there is a conflict between a package function name and another function used in the same session.

9.12.1. ChiSquare

CALLING SEQUENCE : ChiSquare(F , v)
PARAMETERS : F – ChiSquare value
v – degrees of freedom

- ChiSquare returns the value of x such that:
  int(u^(v/2–1)*exp(–u/2),u=0..x)/(GAMMA(v/2)*(2^(v/2)))=F

EXAMPLES :
ChiSquare(.10,5); ==> 1.610307987
SEE ALSO : help(stats,distribution);

9.12.2. Exponential

CALLING SEQUENCE : Exponential(lambda,value) or Exponential(lambda)
PARAMETERS : lambda – the Exponential lambda value
value – the upper bound of the integral

- If Exponential is just passed a lambda value the procedure will return an unnamed procedure which is the Exponential distribution for that specific lambda value. This procedure then can be evaluated for any given value.
- If the Exponential procedure is passed a lambda value and an actual value the procedure will return the value of the Exponential distribution evaluated to the given bound.

EXAMPLES :
```
temp:=Exponential(5):
temp(4);                ==>   5 exp(-20)
Exponential(5,4);       ==>   5 exp(-20)
```
SEE ALSO : help(stats,Uniform); help(stats,distribution);

9.12.3. Fdist – Variance Ratio Distribution

CALLING SEQUENCE : Fdist(F,n,m)
PARAMETERS : F – percentile
n,m – numerator and denominator degrees of freedom

- This procedure finds an x such that:

$$F = \mathrm{GAMMA}((n+m)/2) / (\mathrm{GAMMA}(n/2)*\mathrm{GAMMA}(m/2)) * \mathrm{int}((n/m*u)^{(n/2-1)} * (1+n/m*u)^{(-(n+m)/2)}, u = 0..x) .$$

EXAMPLES :
```
Fdist(.9,2,5);    ==>   3.779716079
```
SEE ALSO : help(stats,Ftest);

9.12.4. Ftest – test Prob(F>x)

CALLING SEQUENCE : Ftest(x,n,m)
PARAMETERS : x – some value
n,m – numerator and denominator degrees of freedom

- This procedure will evaluate the following integral and take its probabilistic complement (subtract it from 1):

$$F = \mathrm{GAMMA}((n+m)/2) / (\mathrm{GAMMA}(n/2)*\mathrm{GAMMA}(m/2)) * \mathrm{int}((n/m*u)^{(n/2-1)} * (1+n/m*u)^{(-(n+m)/2)}, u = 0..x) .$$

EXAMPLES :
Ftest(32,3,5):
evalf(%); ==> .0010910049
SEE ALSO : help(stats,Fdist)

9.12.5. N – Values for a Normal Distribution

CALLING SEQUENCE : N(x,m,v);
PARAMETERS : x – upper bound of integration
m – <optional> mean of the distribution
v – <optional> variance of the distribution

- N integrates the following function:

  N = int(exp(−1/2*(u−m)^2/v) / sqrt(2*Pi*v), u = −infinity..x) .

- The default values are m=0 and v=1 (i.e., the N(0,1) distribution). If m and v are specified then the distribution is N(m,v).

EXAMPLES :
N(2.44); ==> .9926563690
N(2.44,2,5); ==> .5779977938
SEE ALSO : help(stats,distribution);

9.12.6. Rsquared – square of the correlation coefficient

CALLING SEQUENCE : Rsquared(list1,list2) or Rsquared(dat,key1,key2)
PARAMETERS : list1,list2 – two lists of values of the same size
dat – a statistical array
key1,key2 – two keys defined in the data

- Rsquared determines the square of the correlation coefficient.
- The Rsquared statistic is the proportion of variance in one of the variables that can be explained by variation in the other variables.

EXAMPLES :
Rsquared([1,2,3,4,5],[3.5,9.7,2.3,10.4,11.66]); ==> .3936148819
dat:=array([[y,x1,x2],[2,5,7],[3,6,8],[4,7,9]]);
Rsquared(dat,x1,x2); ==> 1
SEE ALSO : help(stats,correlation), help(stats,variance)
help(stats,data)

9.12.7. StudentsT – the Student's T distribution

CALLING SEQUENCE : StudentsT(F,v)
PARAMETERS : F – cumulative probability
v – degrees of freedom

- StudentsT returns the x value to satisfy the following integral:

F = GAMMA((v+1)/2) / (sqrt(Pi*v)*GAMMA(v/2)) *
int((1+u^2/v)^(–(v+1)/2), u = –infinity..x) .

EXAMPLES :
StudentsT(.75,4); ==> .7406970841;
SEE ALSO : help(stats,distribution);

9.12.8. Uniform – the Uniform distribution

CALLING SEQUENCE : Uniform(a,b)
PARAMETERS : a,b – the range of the Uniform distribution

- A Uniform distribution has value 0 everywhere except a..b where its value is 1/(b–a).
- The procedure Uniform returns an unnamed procedure which is the Uniform distribution of the given interval.

EXAMPLES :
k:=Uniform(a,b):
k(); ==> {[1/(b–a),a..b]}
j:=Uniform(5,7):
j(6); ==> 1/2
j(10); ==> 0
SEE ALSO : help(stats,Exponential); help(stats,distribution);

9.12.9. addrecord – add records to a statistical matrix

CALLING SEQUENCE : addrecord(dat,new);
PARAMETERS : dat – a statistical matrix
new – a matrix or list of values

- Addrecord appends new observations to a statistical matrix. The new statistical matrix is returned.

EXAMPLES :
dat:=array([[x,y],[1,2],[2,3]]):
addrecord(dat,[3,4]); ==> array([[x,y],[1,2],[2,3],[3,4]]):
SEE ALSO : help(stats,getkey); help(stats,removekey); help(stats,putkey);

9.12.10. average – calculate the average of some numbers

CALLING SEQUENCE : average(x) or average(x1,x2,x3,...)
or average(dat,k1,k2,...);
PARAMETERS : x – a list,list of lists,matrix, or array
x1,x2,... – sequence to be averaged
dat – statistical matrix
k1,k2,... – keys of statistical matrix

- Average takes the average of some numbers.
- If average receives a matrix, list of lists, or a statistical matrix with more than one key specified average will return a list of values corresponding to the averages of the columns

EXAMPLES :
```
k:=array([[1,2,2],[5,5,5],[11,7,8]]):
average(k);                          ==>  [17/3,14/3,5]
average(");                          ==>  46/9
```
SEE ALSO : help(stats,variance); help(stats,mode); help(stats,median);

9.12.11. correlation – evaluate the correlation coefficient

CALLING SEQUENCE : correlation(dat,key1,key2) or correlation(list1,list2)
PARAMETERS : dat – standard statistical matrix
key1,key2 – two keys from dat
list1,list2 – two lists of values

- The correlation coefficient shows how strong a relationship exists between two continuous variables.

EXAMPLES :
```
correlation([1,2,3,4,5,6,7],[1,2,5,8,10,13,16]):
evalf(");                                    ==>  .9954022746
```
SEE ALSO : help(stats,Rsquared), help(stats,covariance)

9.12.12. covariance

CALLING SEQUENCE : covariance(dat,key1,key2) or covariance(list1,list2)
PARAMETERS : dat – statistical matrix
key1,key2 – two keys from dat
list1,list2 – two lists of values

- The covariance between two lists of data.

- A single number representing the covariance will be returned.

EXAMPLES :
```
covariance([1,2,3],[1.2,2.3,3.7]);                    ==>   1.25
dat:=array([[x,y,z],[1,7,1.2],[2,10,2.3],[3,20,3.7]]):
covariance(dat,x,z);                                   ==>   1.25
```
SEE ALSO : help(stats,variance)

9.12.13. evalstat – manipulation of statistical data

CALLING SEQUENCE : evalstat(dat,eqn1,eqn2,...)
PARAMETERS : dat – basic statistical matrix
eqn1,eqn2 – new equations involving current data

- Evalstat constructs a new column in the statistical array with values determined by the given equation.
- This procedure returns the new array which can be assigned to another variable name.

EXAMPLES :
```
dat:=array([[x,y],[1,2],[3,4]]):
evalstat(dat,k=5*x+y^2);        ==>   array([[x,y,k],[1,2,9],[3,4,31]])
```
SEE ALSO : help(stats,data);

9.12.14. getkey – returns the key from a statistical matrix

CALLING SEQUENCE : getkey(dat);
PARAMETERS : dat – statistical matrix

- Getkey returns a list which is the key of the given statistical matrix.

EXAMPLES :
```
dat:=array([[x,y,z],[1,2,3],...]):
getkey(dat);                          ==>   [x,y,z]
```
SEE ALSO : help(stats,putkey), help(stats,removekey)

9.12.15. linregress – linear regression

CALLING SEQUENCE : linregress(dat,eqn) or linregress(yvals,xvals);
PARAMETERS : dat – standard statistical matrix
eqn – relation of variables
yvals,xvals – lists of values

- This routine uses the traditional estimate Sxy/Sxx=b for the equation y=a+b*x. The "a" value is estimated by a=ybar–b*xbar.

- The parameter "eqn" must be of the form key1=key2 which are two of the keys in the array.
- The procedure returns a list containing the intercept value and the slope value.

EXAMPLES :
linregress(dat,y=x5); ==> [a,b]
linregress([3,4,5,6],[10,15,17,19]); ==> [−.4413407821, .3240223464]
SEE ALSO :help(stats,multregress), help(stats,regression)

9.12.16. median – the median of the given data

CALLING SEQUENCE : median(data) or median(dat,k1,k2,...)
PARAMETERS : data – either a list, matrix, list of lists, or a bunch of numbers
dat – statistical matrix
k1,k2,... – keys of the statistical matrix

- The median of a list of n numbers is the round(n/2) number when the list is sorted.
- If a list of list, matrix, or statistical matrix with more than one key is specified then median will return a list of values.

EXAMPLES :
median(3,4,5,6,7); ==> 5
dat:=array([[1,2,3,4],[5,6,7,8],[9,2,4,1]]);
median(dat); ==> [5,2,4,4]
SEE ALSO : help(stats,average); help(stats,mode);

9.12.17. mode – the mode of the given data

CALLING SEQUENCE : mode(data,lowbnd,incr);
PARAMETERS : data – either a list, matrix, list of lists, or a bunch of numbers
lowbnd – <optional> the lower bound of the data
incr – <optional> the desired range increment

- The mode of a set of n numbers is the number(s) with the highest frequency in the data set.
- If only "data" is specified then each unique number is considered an observation.
- If a lower bound and a range increment are specified then the range of numbers with the maximum frequency of observations is considered the mode. The lower bound must be less than the minimum of the data or

erroneous results will occur.

- If more than one number or range has the highest frequency then a list of modes will be returned.
- If the data specified is a matrix or a list of lists then a list of lists will be returned containing the lists of modes for each column.

EXAMPLES :
```
dat:=[1,1,2,2,2,2,5,5,9,9,9,9];
mode(dat);                                  ==>   [2,9]
mode(dat,1,2);                              ==>   [1..3]
dat:=array([[1,2,3,4],[1,2,3,3],[1,2,3,2]]);
mode(dat);                                  ==>   [[1],[2],[3],[2,3,4]]
```
SEE ALSO : help(stats,average); help(stats,median);

9.12.18. multregress – multiple regression

CALLING SEQUENCE : multregress(dat,eqn,const);
PARAMETERS : dat – standard statistical array
eqn – relation of the form y=[x1,x2,...]
const – <optional> constant term will be added

- Multregress uses the matrix equation:

$$(X^t X)^{-1} X^t Y$$

- An array containing the evaluated coefficients is returned.
- The form of the equation should be y=[x1,x2,x3,...] where y,x1,x2,x3,... are keys of the statistical matrix. A list of the coefficients for each of x1,x2,etc. is calculated
- If the option "const" is specified as the third parameter then a constant term is added as the first entry in the return array.

EXAMPLES :
```
multregress(dat,x5=[x2,x4,x6],const);
 returns array([a0,c2,c4,c6]);
 where x5=a0+c2*x2+c4*x4+c6*x6 is the regression equation.
```
SEE ALSO : help(stats,regression), help(stats,projection)
help(stats,linregress)

9.12.19. projection – create a projection matrix

CALLING SEQUENCE : projection(dat,eqn,const);

PARAMETERS : dat – statistical matrix
eqn – relation of the form y=[x1,x2,x3,...]
const – <optional> constant term will be added

- This procedure constructs a projection matrix according to:

$$X\,(X^t\,X)^{-1}\,X^t$$

- The equation must be of the form y=[x1,x2,...]
- The projection matrix projects the actual Y values onto the predicted Y values.
- If the option "const" is added as a third parameter then the projection matrix will be constructed as if a constant existed in the equation.

EXAMPLES :
pro:=projection(dat,y=[x1,x4,x5],const);
SEE ALSO : help(stats,regression)

9.12.20. putkey – add a key to a matrix

CALLING SEQUENCE : putkey(dat,key);
PARAMETERS : dat – matrix full of data
key – list of names making up the key for the data

- This procedure adds a row to the given data matrix making it a statistical matrix.
- The key must be a list of syntactically correct names.
- This procedure returns a new unnamed array with the key added.

EXAMPLES :
dat:=array([[2,3],[5,7]]):
putkey(dat,[y,x]); ==> array([[y,x],[2,3],[5,7]])
SEE ALSO : help(stats,data); help(stats,removekey);

9.12.21. regression – generalized regression routine

CALLING SEQUENCE : regression(dat,eqn)
PARAMETERS : dat – statistical array
eqn – equation for regression fit

- This procedure accepts data and an equation linear in the unknown coefficients and determines values for the coefficients.
- This routine solves the system of normal equations to determine the values of the coefficients.

EXAMPLES :
regression(dat,y=a+b*x+c*x^2+d*ln(x4));
returns {a= value, b= value, c= value, etc...}
SEE ALSO : help(stats,multregress), help(stats,linregress)

9.12.22. removekey – take the key away from a statistical matrix

CALLING SEQUENCE : removekey(dat);
PARAMETERS : dat – statistical matrix

- Removekey removes the first row of a statistical matrix and returns a normal matrix full of the data.

EXAMPLES :
dat:=array([[y,x],[3,4],[6,7],[7,8]]):
removekey(dat); ==> array([[3,4],[6,7],[7,8]])
SEE ALSO : help(stats,putkey); help(stats,getkey); help(stats,data);

9.12.23. sdev – standard deviation

CALLING SEQUENCE : sdev(x) or sdev(dat,k1,k2,...)
PARAMETERS : x – either a list, list of lists, matrix, or vector
dat – statistical matrix
k1,k2,... – keys of the statistical matrix

- This procedure calculates the standard deviation of a list of numbers.
- Standard deviation is a measure of how widely the numbers are spread.
- If a list of lists,matrix,or a statistical matrix with more than one key is specified then a list of values is returned.

EXAMPLES :
sdev(3,4,5,6,7,8,100); ==> 2*sqrt(959)/sqrt(3)
dat:=array([[y,x],[3,4],[9.5,6.7],[0.001,0.005]]):
sdev(dat,y,x); ==> [4.855838445, 3.368309417]
SEE ALSO : help(stats,var); help(stats,serr);

9.12.24. serr – standard error

CALLING SEQUENCE : serr(x) or serr(dat,k1,k2,...);
PARAMETERS : x – can be list, list of lists, matrix, or vector
dat – statistical matrix
k1,k2,... – keys of the statistical matrix

- The standard error of a list of numbers tells us how far off the mean estimate might be.
- If a list of lists, matrix, or a statistical matrix with more than one key is specified then a list of values is returned.

EXAMPLES :
```
serr([3,4,0.001,9.9]);    ==>    2.073625062
dat:=array([[x,y],[2,3],[6.7,31],[0.001,0.005]]):
serr(dat,x,y);            ==>    [3.439055153/sqrt(3), 17.09610116/sqrt(3)]
```
SEE ALSO : help(stats,sdev); help(stats,variance);

9.12.25. statplot – statistical plotting

CALLING SEQUENCE : statplot(dat,eqn or prod)
PARAMETERS : dat – statistical array
eqn or prod – an equation or a product of the keys

- A specially designed plotting routine to help with statistical plotting
- If an equation is specified as the second parameter then the line is plotted on top of the x and y points. This is very handy in plotting regression results.
- If a product is specified then this is interpreted as "x vs y"
- The plotdevice must be changed outside of this routine.

EXAMPLES :
```
statplot(dat,y*x);      – see if linear regression is appropriate
  results in a plot of y vs x
regression(dat,y=A+B*x); – where y,x are keys to dat
statplot(dat,y=A+B*x);  – regression assigns the calculated
                          values to A,B
  results in a plot of y vs x with the regression line
  overlaid.
```
SEE ALSO : help(stats,data); help(stats,regression);

9.12.26. variance

CALLING SEQUENCE : variance(x) or variance(dat,k1,k2,...)
PARAMETERS : x – can be a list, matrix, or vector
dat – a statistical matrix
k1,k2,... – keys of the statistical matrix

- The variance is a measure of spread in the data.

- If a list of lists, matrix, or a statistical matrix with more than one key is specified a list of values is returned.

EXAMPLES :
variance([3,4,5,6]); ==> 5/3
dat:=array([[x,y],[3,5],[6.7,8.9],[0.001,0.005]]):
variance(dat,y,x); ==> [19.88017500, 11.26010033]
SEE ALSO : help(stats,covariance); help(stats,sdev);

9.13. The `student` package for Calculus

SYNTAX : Load as with(student);

- The student package is a collection of routines designed to carry out step by step solutions to problems. Integrals, Sums and Limits are handled in an unevaluated form. Routines for expanding or combining such expressions or for performing change variables, integration by parts, completing the square or replacing factors of expressions by names are included. Other routines compute distances, slopes, and find midpoints of line segments, or provide information about graphs of functions.
- The routines used include:

changevar	cmpltsq	combine	D	diff
distance	Eval	Int	intercept	intparts
isolate	Limit	makeproc	midpoint	Point
powsubs	slope	Sum		

- In the event of name conflicts, use the long form student[fname](args);
- The routines in this package have been used to prepare the document "Teaching First Year Calculus through the Use of Symbolic Algebra" by J. S. Devitt.

SEE ALSO : help(student,<function>);

9.13.1. D – Differential operator

CALLING SEQUENCE : D(f)

PARAMETERS :

f A maple expression which can be applied as a function.

- D(f) is a function defined so that D(f)(x) = diff(f(x),x). That is, D(f) = unapply(diff(f(x),x),x).
- The argument f must be an algebraic expression which can be treated as if it were a function. All names are treated as function names. For example, f= g+h and f = f*h are valid since (g+h)(x) = g(x) + h(x) and (g*h)(x) = g(x)*h(x).
- The usual rules for differentiation hold including, for example, D(sin) = cos and D(1/f) = –D(f)/f^2 . Functional composition is represented by g @ h.

EXAMPLES :

D(f) yields D(f)
D(f*g); yields D(f) g + f D(g)

D(f@g); yields (D(f) @ g) D(g)
D(y^2=x^3 +x); yields 2 D(y) y = 3 D(x) x^2 + D(x)
SEE ALSO : diff, `@`, `@@`, unapply, help(operators,example)

9.13.2. Eval – evaluate inert functions

CALLING SEQUENCE : Eval(f)
PARAMETERS :

f any algebraic expression.

- Maple is moving towards a policy of using function names which begin with a capital letter for unevaluated forms of the corresponding functions whose names are all in lower case.
- Eval() forces `unevaluated` functions such as Int(), Limit(), and Int() to be evaluated by their corresponding maple function (int(), limit(), or sum()). This is easily done by substituting upper case names by their lower case equivalents and invoking eval().
- Only Int, Limit, and Sum, are known to Eval() at this time.

EXAMPLES :
Eval(Int(x,x)); yields x^2/2
Eval(Limit(sin(x)/x,x=0); yields 1
Eval(Sum(i,i=1..n) + 5); yields 1/2*(n+1)**2-1/2*n+9/2;
SEE ALSO : int, limit, sum, student, eval

9.13.3. Int – inert form of int (integration function)

CALLING SEQUENCE : Int (f, x = a ... b) or Int(f,x)
PARAMETERS :

f	The expression in x which is to be integrated.
x	The variable used to perform the integration.
a and b	Optional upper and lower bounds defining the range of integration.

- This function corresponds to the integral of f with respect to x. It is an `unevaluated` form of Maple's int() function so only minor simplifications are performed. Such unevaluated expressions can be manipulated by routines in the `student` package such as changevar(), intparts(), combine(), and expand(). Partial fraction decompositions are available through convert(...,parfrac).
- Use Eval() to force Int() to evaluate like int().

EXAMPLES :
Int(f(x),x);
Int(f*g,x=1..n);
SEE ALSO : int, student, int, Eval, changevar, intparts, parfrac

9.13.4. Limit – inert form of limit

CALLING SEQUENCE : Limit (f , x = a) Limit(f , x = a , right);
PARAMETERS :

f	The expression in 'x'.
a	The final value of x.
right	optional parameter (also real, left, and complex)

- This function corresponds to the limit of f as x approaches a. It is an `unevaluated` form of Maple's limit() function so only minor simplifications are performed. It can be manipulated by changevar() and expand(). The combine() function can be used to rewrite sums and products as one Limit().
- Use Eval() to force evaluation by the limit() function.

EXAMPLES :
Limit(x^3 + 3*x^2 + x + 1,x=4);
Limit(1/x,x=0,right); Limit(1/x,x=0,real);
expand(Limit(x^3 + 3*x^2 + x + 1,x=4)); yields
Limit(x,x = 4)**3+3*Limit(x,x = 4)**2+Limit(x,x = 4)+1
Eval(") yields 117
SEE ALSO : limit, student, Int, Sum, Eval, changevar , combine

9.13.5. type/Point – check for type Point

CALLING SEQUENCE : type(P,Point)
PARAMETERS : P any algebraic expression

- A Point is defined as a list of coordinate values (e.g. [a,b,c]). The length of the list determines the dimension of the space.
- They differ from objects of maple type `point` (e.g. {x=a,y=b,z=c}) in that the coordinates are defined by position in the list rather than by names. subs({x=a,y=b,z=c},[x,y,z]) converts a `Point` to a `point`.

EXAMPLE :
type(a , Point); yields false
type([a], Point); yields true (one dimension)
type([a+4,b] , Point); yields true (two dimensions)
SEE ALSO : student, distance, midpoint, slope

9.13.6. Sum – inert form of sum

CALLING SEQUENCE : Sum (f(i), i = a ... b)
PARAMETERS :

f	The summand (an expression in i)
a and b	Optional upper and lower bounds defining the range of summation for i.

- This function corresponds to the sum of f(i) for i in the given range. It is an unevaluated form of the maple sum() function so only some minor simplifications are performed. Expressions involving Sum() can be manipulated by routines such as expand(), combine() and changevar().
- Use Eval() to force Sum() to evaluate like sum().

EXAMPLES :
```
Sum(f(i),i=a..b);
Sum(x^2+x,x=1..n);
expand(%);          yields  Sum(x^2,x=1..n) + Sum(x,x=1..n)
```
SEE ALSO : sum, student, combine, expand, Eval, changevar

9.13.7. changevar – perform a change of variables

CALLING SEQUENCE : changevar (s, f, u);
PARAMETERS :

s	an expression f(x)=g(u), defining x as a function of u.
f	an expression such as Int(F(x),x=a...b)
u	the name of the new integration (summation) variable.

- Execute a `change of variables` for integrals, sums, or limits. The first argument is an equation defining the new variable in terms of the old variable. If more than two variables are involved, the new variable must be given as the third argument. The second argument is the expression to be rewritten and usually contains Int, Sum, or Limit.
- Changevar() acts like powsubs() if none of Int, Sum, or Limit appears.
- The change of variables may be implicitly defined (e.g. x^2+2=2*u^2).
- You should use the unevaluated forms Int(), Limit(), and Sum() rather than int(), limit(), and sum(). They can be evaluated later by Eval().

EXAMPLES :
```
changevar( cos(x)+1=u,Int((cos(x)+1)^3*sin(x),x),u);
        Int(-u**3,u)
changevar( x=sin(u),Int(sqrt(1-x^2),x=a...b),u);
    Int((1-sin(u)**2)**(1/2)*cos(u),u = arcsin(a) .. arcsin(b))
```

SEE ALSO : student, powsubs, subs, Int, Limit, Sum

9.13.8. cmpltsq – complete the square

CALLING SEQUENCE : cmpltsq (f), cmpltsq(f , x)
PARAMETERS :

f an algebraic expression
x one of the indeterminates occurring in f.

- This function completes the square of a polynomials of degree 2 in x by rewriting such polynomials as perfect squares plus a remainder. If more than one variable appears in f, then x must be specified.
- The polynomial may appear as a subexpression of f, so that equations, reciprocals, and in general, parts of expressions may be simplified. As the polynomial may occur as an argument to Int() this can be used to help simplify unevaluated integrals.
- Terms not involving the indicated variable are treated as constants.

EXAMPLES :
cmpltsq(9*x^2+24*x+16) yields 9*(x+4/3)**2
cmpltsq(x^2–2*x*a+a^2 + y^2–2*y*b+b^2 = 23,x) yields
(x–a)^2+y^2–2*y*b+b^2 = 23
cmpltsq(",y) yields (x–a)^2+(y–b)^2 = 23
SEE ALSO : student, Int

9.13.9. combine – combine terms into a single term

CALLING SEQUENCE : combine (f)
PARAMETERS :

f any algebraic expression

- Undo the effect of expand() on unevaluated expressions such as Int(), Sum(), and Limit(). The argument may be any algebraic expression. Subexpressions involving Int(), Sum(), and Limit() are combined into one expression where possible using linearity. I.e., c1*f(a,range) + c2*f(b,range) = f(c1*a+c2*d,range).
- Limits are allowed to combine under products and powers.
- Checks are made to ensure that all arguments beyond the first of Int(), Sum(), or Limit() are identical.

EXAMPLES :
combine(Limit(x,x=a) * Limit(x^2,x=a)+c) yields
Limit(x^3+c,x=a) ;
combine(Int(x,x=a..b) – Int(x^2,x=a..b)) yields

```
    Int( x-x^2 , x = a...b );
combine(a+b) yields   a+b
```
SEE ALSO : student, Int, Sum, Limit, expand

9.13.10. diff – differentiate with respect to specified variable(s)

CALLING SEQUENCE : diff(a,x), diff(a, x1, x2, ..., xn);
PARAMETERS : a – any algebraic expression
x1, ... – names

- "diff" computes the partial derivative of "a" with respect to the variables x1, x2, ..., xn, respectively.
- The call "diff(f(x),x,y);" is equivalent to the call "diff(diff(f(x),x),y);"

EXAMPLES :
diff(sin(x),x); yields cos(x)
diff(sin(x),y); yields 0
diff(1/(x+2),x); yields -1/(x+2)^2
SEE ALSO : help(int), help(dsolve), help(diff)

9.13.11. distance – compute the distance between Points

CALLING SEQUENCE : distance (a,b)
PARAMETERS : a,b expressions or objects of type Point

- Compute the distance between Points in one or higher dimensions. If the Points are dimension 2 or higher (e.g. [a,b] or [1,2]) then the standard Euclidean distance is used. If one or both Points are algebraic expressions other than names, then the distance is computed in one dimension. If the Points are both names then distance returns unevaluated.
- A Point in two dimensions is defined as a list of coordinates (e.g. [a,b]). Points in one dimension are just algebraic expressions, though names may be treated as either depending on the context.

EXAMPLE :
distance(a,b); yields distance(a,b)
distance(-3,a+5); yields abs(a+8)
distance([a,b],[c,d]); yields (a^2-2*a*c+c^2+b^2-2*b*d+d^2)^(1/2)
SEE ALSO : student, midpoint, slope, cmpltsq, Point

9.13.12. intercept – compute the points of intersection of two curves

CALLING SEQUENCE : intercept (eq1);
intercept (eq1 , eq2 , {x,y});

PARAMETERS :

eq1,eq2	bivariate equations (e.g. y = x^2 + 3)
x,y	(optional) The coordinate variables.

- Compute the points corresponding to the intersection of the two curves defined by eq1 and eq2.
- If only one equation is given, the "y" intercept (i.e. the intercept with x=0) is returned. In this case, the dependent variable should appear alone on the left hand side of the given equation as in "y=3*x + 5".
- An optional third argument is used to indicate the set of coordinate variables.

EXAMPLES :

```
intercept(y=x+5);              yields {x=0,y=5}
intercept(y=x+5,y=0);          yields {x=-5,y=0}
intercept(y+4=x+5,x=0);        yields {x=0,y=1}
intercept(y=a*x+b,x=0,{x,y}); yields {x=0,y=b}
```

SEE ALSO : student, intercept, midpoint

9.13.13. intparts – perform integration by parts

CALLING SEQUENCE : intparts (f,u)

PARAMETERS :

f	An expression of the form Int(u*dv,x)
u	The factor of the integrand to be differentiated.

- Carry out integration by parts on an unevaluated integral (i.e. expressed in terms of Int).
- The function returns u*v – Int(du*v,x) as its value.

EXAMPLES :

```
intparts(Int(x^k*ln(x),x),ln(x));
intparts(Int(sin(x)*x+sin(x),x),sin(x));
```

SEE ALSO : student, powsubs, changevar

9.13.14. isolate – isolate a subexpression to left-side of an equation

CALLING SEQUENCE : isolate(eqn,expr); or isolate(eqn,expr,iter);

PARAMETERS : eqn – an equation or an algebraic expression
expr – any algebraic expression
iter – an (optional) positive integer

- The procedure "isolate" attempts to isolate the second argument "expr" in the first argument "eqn" and solves for "expr" in "eqn" .
- If the first argument is not an equation, then the equation eqn=0 is assumed.
- The optional third argument "iter" controls the maximum number of transformation steps that "isolate" will perform.

EXAMPLES :

```
isolate(4*x*sin(x)=3,sin(x));   yields  sin(x) = 3/4/x
isolate(x^2-3*x-5,x^2);         yields  x^2 = 3*x+5
```

SEE ALSO : help(isolate), help(solve)

9.13.15. makeproc – converts an expression into a maple procedure

CALLING SEQUENCE : makeproc(expr,x)
PARAMETERS : expr – any expression
x – variable name

- The result of makeproc(expr,x) is a procedure which when evaluated at x returns "expr".
- It is useful when evaluating an expression at several values of x.
- If "expr" is an unevaluated function call such as sin(x), then the name of the function is returned.
- Makeproc should be used instead of the < > operator notation to construct a Maple procedure from the value of an expression used in your current session. The < > notation also defines a Maple procedure, but requires that you type in the entire defining expression in its final form.
- Makeproc is loaded via the command "with(student)".

EXAMPLES :

```
makeproc(sin(x),x)  yields  sin
p:= x^2 + 2*x + 3;
f := makeproc(p,x);
f(x);               yields  x^2 + 2*x + 3
f(u);               yields  u^2 + 2*u + 3
```

SEE ALSO : unapply, student, operators

9.13.16. midpoint – compute midpoint of a line segment

CALLING SEQUENCE : midpoint (p1, p2)
PARAMETERS :

p1,p2 names, expressions or points.

- Compute the midpoint of the line segment defined by the two points p1 and p2. The points are either lists of length 2, or expressions. In the latter case, distance is just their average.
- If one argument is a point and the other is a name, then the name is treated as an unevaluated point.

EXAMPLES :

```
midpoint( [a,b],[c,d]); yields [(a+c)/2, (b+d)/2]
midpoint( –5, 8);       yields 13/2
midpoint( c, [a,b] );   yields [(a–c[1])/2 , (b–c[2])/2 );
```

SEE ALSO : student slope

9.13.17. powsubs – substitute for factors of an expression

CALLING SEQUENCE : powsubs (eqn, f)
PARAMETERS :

eqn An equation s1=s2 specifying the substitution
f An expression containing the lefthand side of eqn as An algebraic factor of a subexpression of f.

- powsubs() differs from subs in that it is defined in terms of algebraic factors rather than in terms of an underlying data structure. Each subexpression is examined and if s1 occurs as a factor of that subexpression, the substitution is carried out. powsubs() cannot find expressions which only occur as part of a sum.
- If more than one substitution is specified they are processed in order left to right. Powsubs does not carry out simultaneous substitutions.

EXAMPLES :

```
subs( a^2*c=b, a^2*c*d );          yields a^2*c*d
powsubs( a^2*c=b, a^2*c*d );       yields b*d
powsubs( x=y^2, x*sin(x) );        yields y^2*sin(y^2)
powsubs( x^2=b, x^3+x^5 );         yields x*b + x*b^2
powsubs( x^2=b, x=4, x^3+x^5 );    yields 4*b + 4*b^2
```

SEE ALSO : subs, student

9.13.18. slope – compute the slope of a line

CALLING SEQUENCE : slope(equation)
slope(equation,y,x), slope(equation,y(x))
slope(p1,p2)

PARAMETERS :

equation	An equation of the form y=a*x+b.
y,x	(optional in most cases) the dependent and independent variable.
p1,p2	Two 2–dimensional points (i.e. [a,b] and [c,d])

- Compute the slope of a line as determined by two points, or an equation.
- For equations of the form, y=f(x), the dependent variable is taken to be "y". Ambiguity (as a result of 3 or more variables or use of a different form of equation) is resolved by providing two additional arguments, "y" and "x", the names of the dependent and independent variables.

EXAMPLES :

```
slope(y=3*x+5);          yields 3
slope(y=a*x+5,y,x);      yields a
slope(2*y+4=x–3,x,y);    yields 2
slope(2*y+4=x–3,x(y));   yields 2
```

SEE ALSO : student, intersect, midpoint

10. MISCELLANEOUS FACILITIES

10.1. Debugging facilities: Detecting syntax errors

10.1.1. `Mint`, the Maple program diagnostician

The standard mode for detecting syntax errors in Maple procedure definitions is to develop the procedure definition into a file (using a text editor external to Maple) and then to use the **Mint** facility to detect any syntax errors. Mint is a program distinct from Maple. It inspects a specified file (assumed to be a Maple source file) and produces a report on syntax errors encountered and on how variables are used.

The precise command syntax for Mint is system-dependent. (See Appendices A, B, and C for system-specific details.) Generally, Mint processes a specified input file and produces a report. The level of detail presented in the report from Mint can be controlled by an *information level* flag, which takes an integer value between 0 and 4 with the following meanings:

0 – display no information
1 – display only severe errors
2 – display severe and serious errors
3 – display warnings as well as severe and serious errors
4 – give a full report on variable usage as well as displaying errors and warnings

The definitions of the various categories of errors and warnings is given below. By default, Mint will use information level 2. A report for each procedure in the input file is displayed separately, followed by a global report for statements not contained within any procedure.

The types of errors and warnings reported are classified as severe errors, serious errors, and warnings. A severe error is an indisputable error. A serious error is almost certainly an error, but in some circumstances the programmer may choose to ignore it. Warnings are possible errors. They point to constructs that may be correct in some contexts, but are probably errors in other contexts. More detailed definitions of these categories follow.

Severe Errors

- *Syntax errors*: A caret symbol (or some other system-dependent symbol) will point to the token that was being read when the error occurred.

- *Duplicated parameter*: A name appears more than once in a parameter list for a procedure.
- *Duplicated local*: A name is declared more than once in the list of local variables for a procedure.
- *Local variable and parameter conflict*: A name is used both as a parameter and as a local variable within a procedure. In further analysis, the name is treated as a parameter.
- *Local variable and system-defined name conflict*: The name of a local variable is also used by Maple as a system-defined name.
- *Parameter and system-defined name conflict*: The name of a parameter is also used by Maple as a system-defined name.
- *Duplicated loop name*: A loop nested within another loop uses as its loop control variable the same name that the outer loop uses.
- *Break or next statement outside of a loop*: A break or a next statement occurs outside of any loop. (Break or next may still be used as names within an expression outside of a loop.)
- *RETURN or ERROR function call outside of a procedure*: A function call to RETURN or ERROR occurs outside of a procedure body. (RETURN or ERROR may still be used as names if they are not invoked as functions.)
- *Unreachable code*: There are statements which follow directly after a "goto" type of statement. These statements are unreachable and will never be executed. A "goto" statement is a **next** statement, a **break** statement, a **quit**, **stop**, or **done** statement, a RETURN() call, or an ERROR() call. An **if** statement all branches of which end in a goto statement is also considered a goto statement.

Serious Errors

- *Overly long name*: A name whose length is too long is used. The name is truncated to the maximum allowed.
- *Unused local variable*: A local variable is declared for a procedure but never used within the procedure body.
- *Local variable assigned a value but not used otherwise*: A local variable is assigned a value within a procedure but is not otherwise used.
- *Local variable never assigned a value but used as a value*: A local variable was never assigned a value in a procedure but within the procedure its value is used in an expression. Such an expression would contain a pointer to a non-existent local variable if the expression were returned or assigned to a global variable.

- *System-defined name is overwritten*: A name which is treated as a system-defined name by Maple is assigned a value. Included in the report is an indication of which parts of Maple use the system-defined name.
- *Dubious global name in a procedure*: A global name is used within a procedure. A global name is a name which is not a parameter, a local variable, a system-defined name, nor a concatenated name. Global names used as procedure names in a function call are not considered errors, nor are names beginning with '_'.
- *Library file name overwritten*: The name of a library file, e.g., `convert/ratpoly`, is assigned a value. It is usual for the name of a library file to also be the name of a library function. Hence, the library function would be no longer accessible.
- *Unused parameter in a procedure*: A name specified in the parameter list of a procedure is never used in the procedure. This is considered a serious error if 'args' is never used in the procedure either. If 'args' is used in the procedure, then it is possible that the parameter may be accessed (e.g. args[i] accesses the *ith* parameter) and this error is down-graded to a warning.
- *Wrong argument count in a procedure call*: The number of arguments passed in a procedure call doesn't match the number of formal arguments in the definition of a procedure of the same name recorded in the library database file. The library database file contains information about the minimum number of arguments and the maximum number of arguments expected for a procedure.

Warnings

- *Equation used as a statement*: This may be intentional. On the other hand, it is common for many Fortran and C programmers to mistype '=' for the assignment operator which is ':=' in Maple.
- *Unused parameter in a procedure*: See similar entry under serious errors.
- *Global name used*: A global name which may or may not start with '_' is used within the procedure.
- *Concatenated name used*: A name is formed through the concatenation operator.

Other reports

If the information level is 4, then a usage report is given for each procedure as well as global statements within the file. Each usage report shows how parameters, local variables, global variables, system-defined names, and concatenated names are used. As well as can easily be done, the following information about how a variable is used may be provided:

1. Used as a value.
2. Used as a table or list element.
3. Used as a call-by-value parameter.
4. Used as a call-by-name parameter (a quoted parameter).
5. Called as a function.
6. Assigned a procedure.
7. Assigned a list.
8. Assigned a set.
9. Assigned a range.
10. Assigned a value as a table or list element.
11. Assigned a value to remember as a function value.

In addition, a list of all error messages generated is given.

For details on the precise command syntax for Mint on specific host operating systems, see Appendices A, B, and C.

10.2. Debugging facilities: Monitoring run-time execution

10.2.1. The `printlevel` facility

The *printlevel* facility allows the user to monitor the execution of statements, to selected levels of nesting within procedures. The default value for printlevel is 1. Any *word-size* integer (on typical host computers, the size limit is 9 or 10 digits in length) may be assigned to the name printlevel and, in general, higher values of printlevel cause more information to be displayed. Negative values indicate that no information is to be displayed.

More specifically, there are *levels* of statements recognized within a particular procedure (or in the main session) determined by the nesting of conditional and/or repetition statements. If the user assigns

```
printlevel := 0;
```

then the following statements within the main session

```
b := 2;
for i to 5 do a.i := b^i od;
```

will generate the printout b := 2 after execution of the first statement and

there will be no printout caused by the for-statement (the value of the for-statement is NULL). If the user assigns

```
printlevel := 1;
```

before the above statements are executed (or equivalently, if no assignment to printlevel has been made) then each statement within the for-statement will be displayed as it is executed (in the same manner as if these statements appeared sequentially in the 'mainstream'), yielding the following printouts for the above statements:

```
b := 2
a1 := 2
a2 := 4
a3 := 8
a4 := 16
a5 := 32
```

The statement b := 2 is considered to be at level 0 while the other assignment statements in this example are at level 1 because they are nested to one level in a repetition statement. If statements are nested to level *i* (where if-clauses as well as do-loops create a nesting of statements within a statement) then the value of printlevel must be *i* if the user wishes to see the results of these statements displayed. Putting it another way, the effective value of *printlevel* is decremented by 1 as execution enters each level of nesting of statements.

More generally, statements are nested to various levels by the nesting of procedure invocations as well. The Maple system decrements the effective value of *printlevel* by 5 upon each entry into a procedure and increments it by 5 upon exit, so that normally (with printlevel = 1) there is no information displayed from statements within a procedure. If the user assigns

```
printlevel := 5;
```

in the main session then statements within procedures called directly from the main session (but not nested statements) will be displayed as they are executed, because the effective value of printlevel within the procedure is 0. If the user assigns

```
printlevel := 6;
```

in the main session then, in addition, statements nested to one level in conditional and/or repetition statements in the procedure will be displayed because the effective value of printlevel within the procedure is 1. Alternatively, the user may explicitly set the value of printlevel within the procedure for which the information is desired.

It is often useful for debugging purposes to set a high value of printlevel in the main session if information is desired from within procedures to various levels of nesting. When the effective value of printlevel upon entry to a procedure is 3 or greater, the printout will display the entry point (showing the argument values) and the exit point (showing the returned value) for that procedure. It is not uncommon to use a debug setting such as

```
printlevel := 1000;
```

in which case entry and exit points and statements will be displayed for procedures up to 200 levels deep.

When the value of printlevel is set to 2 or greater at the top (interactive) level, user-information printouts will be displayed from procedures which have been programmed to take advantage of this facility. These printouts typically give information about the methods being attempted and possibly information about the timings of various sub-computations. In the current version of Maple, only a few library procedures have been programmed to give this type of information.

10.2.2. Tracing specific procedures

A more selective debugging tool is provided by the **trace** function. This function is called with the name of a particular procedure. Then, during execution, all entry points (with argument values) and exit points (with returned values) will be displayed for the specified procedure, as well as a trace of each statement executed within the procedure. See the description of **trace** in section 9.1 for further details.

10.3. Monitoring space and time

10.3.1. Run-time messages

As execution proceeds in a Maple session, the user will normally see displayed, at regular intervals, a message of the form:

```
words used=xxxx, alloc=yyyy, time=zzzz
```

where 'xxxx' and 'yyyy' are integers, and 'zzzz' is a floating-point number.

The value 'xxxx' indicates the number of *words used* – i.e., the number of words of memory that have been requested up to that point in the execution of the session. Note that this measure of memory usage is not directly related to the actual memory requirements of the Maple session at any instant, but rather it is a cumulative count of all memory requests made to the internal Maple memory manager during execution of the session.

Typically, a significant proportion of the 'words used' at any time will be re-allocations of actual memory that was previously used and then released to Maple's memory manager. The value 'yyyy' indicates the number of words of memory *actually allocated* for data space during the Maple session. Note that Maple's internal memory manager requests words to be allocated from the host system in large chunks (typically 1024 or 2048 words) and then uses it as needed. The value 'zzzz' is the total CPU time for the session, measured in seconds.

The frequency of the 'words used' messages can be controlled by the user. An implementation of Maple on a particular host system will have a specific *default increment* (such as 100,000 words used) for these messages. The messages will appear at approximately the frequency specified, but the increment between successive messages will be greater whenever a memory-intensive computation is carried out internally (including the case of loading library files). To change the frequency of the messages, the **words** function is called with argument n, where n is an integer specifying the desired increment. A value of 0 is interpreted to mean that the messages should be completely turned off (i.e., an 'infinite' increment). See the **words** function in section 9.1.

The 'words used' message shown above is also displayed whenever a *garbage collection* occurs, even if the words increment has been set to 0. The frequency of garbage collections can also be controlled by the user. See the **gc** function in section 9.1.

Finally, the 'words used' message shown above is displayed when a Maple session is exited via the **quit** (or **done** or **stop**) statement.

10.3.2. The `status` variable

The global variable 'status' may be queried at any time during a Maple session. The value of this variable is an expression sequence of eight numbers, specifying the following quantities:

status[1] – total number of words requested to the storage allocator
status[2] – total number of words allocated
status[3] – number of CPU seconds used
status[4] – words increment for 'words used' messages
status[5] – words increment for automatic garbage collection
status[6] – number of words returned in last garbage collection
status[7] – number of words available after last garbage collection
status[8] – number of times that garbage collection was executed

The first three values in the 'status' sequence are precisely the values appearing in the 'words used' message discussed in the preceding

subsection.

The status variable is updated every time that a 'words used' message is produced and every time that garbage collection occurs. In between these times it remains unchanged.

Example:

```
status;  may yield
     1001684, 288768, 126.417, 100000, 250000, 132847, 140124, 4
status[4];  may yield
                              100000
```

10.3.3. System functions for space and time

Three system functions described in section 9.1 are available for measuring (and controlling) space and time information: **gc**, **words**, and **time**. The **gc** function is used to control the frequency of garbage collections, and also to explicitly invoke a garbage collection process. The **words** function is used to control the frequency of displaying the 'words used' message, and also to query the current value of the 'words used' measure (see the discussion in section 10.3.1). The **time** function is used to query the total CPU time used by the session so far, and thus to measure the CPU time used by any selected sub-computation of a session (by querying **time** before and after the selected sub-computation and taking the difference).

There are two miscellaneous functions described in section 9.2 that may be used to monitor the space and time usage of a computation: **profile** and **showtime**. The **profile** function (with associated functions **showprofile** and **unprofile**) is used to develop a profile of a function (or functions) used in a computation, indicating the number of invocations of the function, the nesting level of each invocation, the CPU time used, and the number of 'words used' by each invocation. The **showtime** function is used to cause the CPU time, and the 'words used', to be displayed for each command entered at the interactive level. It also provides an alternate user interface to Maple in which the prompts are labelled I1, I2, etc., the output values are labelled O1, O2, etc., and previous output values may be accessed by referring to the appropriate O.i value (thus providing a complete history mechanism).

10.4. Global variables

There are ten global variables in Maple whose values may be changed by the user. They are summarized in the following table.

Global variables with user-controlled values		
name	default	description
constants		sequence of symbolic names defined to be of type *constant*; default sequence: false, gamma, infinity, true, Catalan, E, I, Pi
Digits	10	number of digits in floats
log	ln	specifies log as natural logarithm; may be assigned a proc definition
`` `mod` ``	modp	specifies positive representation for modular arithmetic; to use symmetric, assign `` `mod` := mods; ``
Order	6	order of truncation for series
prettyprint	1	0 = linear printing (lprint); 1 = centered prettyprinting; 2 = left-justified prettyprinting
printlevel	1	controls run-time printing (see section 10.2)
prompt	`` `> ` ``	interactive prompt; may be assigned any string
screenheight	24	height of lines for character plots
screenwidth	79	width of lines for printing

In addition to the ten global variables listed above, there are three additional global variables whose values may be queried by the user. The values of these three variables are not normally changed by the user directly but are set by the system. They are summarized in the following table.

Additional global variables	
name	description
lasterror	the expression sequence which is the latest error message generated in the session (see **traperror** function in section 9.1)
libname	pathname of the standard Maple library (implementation dependent)
status	expression sequence specifying system status (see section 10.3.2)

10.5. Maple command line options

The command for invoking the Maple system (**maple** on most installations) can take up to three options (fewer on some installations). Users should consult Appendices A, B, and C for system-specific details.

10.5.1. The Quiet option

This option can be used to start up a Maple session in which no logo will be displayed, no informational messages will be printed, and no sign-off line will be printed. This is particularly useful when the output of a Maple session is to become the input to another process.

10.5.2. The Library Specification option

This option is used to specify the location of the Maple system library. It is typically used by the system installer when installing the Maple system. On most implementations, users can determine the file system pathname of the Maple library by entering a Maple session and displaying the value of the global variable **libname**.

10.5.3. The Suppress Initialization option

There is a command line option which may be specified to suppress the loading of initialization files. The appendices provide system-specific details. The concept of initialization files is as follows.

When a Maple session is begun, the Maple system first searches for a *system initialization file* before it starts receiving any input. The location of this file on a particular host system can be determined by the Maple

command:

convert(``.libname.`/src/init`, hostfile) .

There may be no such file specified for a particular implementation, but if the file exists then it is **read** into the session. This file may contain any valid Maple statements, and it is typically used to set parameters for **gc** and **words**.

Secondly, the Maple system searches for a *user initialization file*. The location of this file varies with different host operating systems – see the appendices for system-specific details. This file may or may not exist for a particular user. Any Maple statements may be placed in this file and these statements are executed before proceeding with the Maple session. It is recommended that the statements in this file be terminated with the silent statement terminator ':' if it is desired to have the initialization proceed silently.

10.6. Other facilities

10.6.1. Escape to host

The character ! when it appears as the first character in a line is treated as an 'escape to host' operator. This allows one to execute commands in the host system from within a Maple session. The line need not terminate with a Maple statement separator (':' or ';') before the carriage-return character terminates the line. Users should consult the appendices for system-specific details.

A. Appendix A – Maple under the DEC VMS operating system

A.1. Maple and VMS

Maple is not designed to be a completely self-contained programming environment; rather, it is designed to work with other services provided by the operating system of the computer on which it runs. Using Maple on VAXes under the DEC VMS operating system requires some knowledge of the fundamentals of VMS, such as logging in and out, using a text editor, and listing and deleting files. This section assumes that the reader knows how to do such activities on VMS without providing further explanation. The reader is referred to the *VMS Primer*[Cor82] for a basic overview of VMS usage.

The standard installation instructions for Maple make it possible for VMS users to start a Maple session by typing MAPLE. This is because Maple should be accessible through the system-wide symbol MAPLE. Additionally, the Maple library will be located by the "system logical" MAPLE$LIB. Another utility which should be available through a system-wide symbol is MINT. Information is available on-line via the VMS HELP command for Maple and Mint.

A.2. Maple initialization files

The file named INIT in the SRC subdirectory of the Maple library plays a special role for all Maple sessions under VMS. If that file exists, it is read (in the sense of the Maple **read** statement) as the first action in any Maple session, before any Maple commands from the user are executed. The INIT file typically contains Maple commands establishing start-up values for Maple system variables, such as those controlled by **gc** and **words**. It is typically set up by the system manager to establish start-up values for such variables that are appropriate for the site.

The "logical-name" MAPLE$INI can be set by a user to the name of a file of the user's choice. When Maple is invoked, this file is read into Maple after the system INIT file (see previous paragraph) is read in, supplementing or overriding the Maple commands found in the INIT file. However, many users find that they don't need a MAPLE$INI file if the default start-up actions work satisfactorily for them.

A.3. Interrupt and Cancel characters

Maple users can employ the usual VMS interrupt character <CTRL/Y> if they wish to quit Maple and return to VMS command level. <CTRL/C> can be used to cancel execution of a Maple calculation but remain in Maple. If <CTRL/C> is typed twice within three seconds, the Maple session will be quit as though a <CTRL/Y> had been typed.

A.4. Temporarily escaping from Maple

When interacting with Maple, typing a line beginning with ! means that what follows is to be interpreted and executed as a VMS command instead of a Maple statement or command. All VMS commands are accessible via this mechanism, e.g., `!edit` or `!show users`. When execution of the VMS command is finished, the Maple session is resumed where it had been suspended. The results of the VMS command do not affect the state of the Maple session when it is resumed.

A.5. Translating between Maple file names and VMS file names

Maple interacts with the VMS file system via the three Maple commands: **save**, **read** and **readlib**. With both **save** and **read**, the user can specify the files to be written or read in the usual VMS-style, e.g., `save a,` `[BWCHAR]FOO` `;`. However, it is possible with **save** and **read**, and obligatory with **readlib**, to use Maple-style file names; e.g., those used in chapters 1-10 of this *Reference Manual*. Maple-style names follow different rules for specifying directories and subdirectories than VMS†.

In the Maple style of naming files, directories (and sub-directories) are separated by /'s when they are included as part of the file name. The part of the name following the last / is interpreted as the file name. File names, like all Maple names, may be created by concatenating several names through the Maple . operator.

There are two kinds of formats for files that Maple reads and writes: source (such as one can create by typing Maple commands as input to a text editor), and internal format (an encoded form that can be more quickly read into a Maple session). On VMS as on other operating systems, Maple follows the rule that if a Maple-style file name ends in `.m` then the file is

† Maple file naming conventions are derived from the Honeywell GCOS and Unix operating systems, where Maple was originally developed. The style has been retained to allow Maple to be easily portable across several different kinds of operating systems.

treated as an internal format file. Otherwise, it is treated as a source file. When a VMS-style name is given in a **read** or **save** statement, it is treated as a source file even if it ends with `.M`.

In the context of a **read**, **save**, or **readlib** command, "Maple-style" file names are automatically translated into the appropriate VMS-style name. As part of this translation process, lower case letters are converted to upper case. Therefore case distinction in file names is lost, with the following exception. If the original "Maple-style" file name (as distinct from sub-directory names that may appear in the path name) begins with an upper case letter, then a prefix is prepended to the resulting VMS file name to maintain this particular case distinction.

Examples:

`read filename;` will read into Maple the VMS file `FILENAME.` in the user's current directory. `FILENAME` is assumed to be in source format.

`save a, `[bwchar]`.`/subdir1/subdir2/`.`foo.m`;` will save the value of the Maple variable `a` in Maple internal format, in the VMS file `[BWCHAR.SUBDIR1.SUBDIR2]FOO.M`.

Maple's **convert** library function can be used to discover how a Maple-style file name will be translated into a VMS-style name. Specifically, `convert(`*Maplefilename*`, hostfile)` will return the VMS name that Maple translates *Maplefilename* into.

Example:

`convert( `[bwchar]/subdir1/subdir2/foo.m`, hostfile)` will return as its value the name `[BWCHAR.SUBDIR1.SUBDIR2]FOO.M`.

A.6. Redirection of input and output

Invoking the `MAPLE` command with < and a VMS-style filename results in the specified file being used as the input source of a Maple session instead of the terminal keyboard. Once all lines have been read from the file, additional input may be entered through the terminal keyboard (unless the session has ended because the file contained a Maple **quit**, **done**, or **stop** command).

Example:

```
MAPLE <IN.DAT   or
MAPLE < IN.DAT
```

results in the contents of IN.DAT being used for input to the Maple session instead of the terminal. This is roughly equivalent to the VMS command sequence

```
DEF/USER SYS$INPUT INPUT.DAT
MAPLE†
```

Similarly, if Maple is invoked with > and a filename, then the results of the Maple session are placed in the specified file instead of being printed at the terminal. Input and output redirection may be used in combination.

While < and > have the same effect as if SYS$INPUT and SYS$OUTPUT were being redefined, these VMS symbols aren't actually redefined.

Examples:

```
MAPLE >FILE.DAT   or
MAPLE > FILE.DAT
```

deposits results of the Maple session in FILE.DAT. It has the same effect as the VMS command sequence

```
DEF/USER SYS$OUTPUT FILE.DAT
MAPLE
```

```
MAPLE < IN.DAT > OUT.DAT
```

takes input from IN.DAT and places results in OUT.DAT. Note that OUT.DAT will not contain the commands (from IN.DAT), but only the results of commands.

---

† If the DEF/USER SYS$INPUT *input_file* approach is taken for Maple input from a file, then *input_file* must contain a **quit** statement as the last line in order to avoid endless re-readings of the file.

A.7. MAPLE command line options for VMS

A.7.1. Overview of Maple command line options

Maple users typically invoke Maple using the command MAPLE. However, Maple can be invoked with up to three options on the command line: *-s*, *-q*, and *-b*‡. They are used to change effects that occur by default during a Maple session, and are explained in the sections that follow.

Command line options can be used together with redirection of input and output from and to files, but then the command line options must appear before the < or >.

Example:

```
MAPLE -q < IN.DAT > OUT.DAT
```

invokes Maple with the *-q* option, taking input from IN.DAT and placing results in OUT.DAT.

A.7.2. Library specification option: -b

The VMS symbol MAPLE$LIB names the default location of the Maple library directory. The command MAPLE `-b` *location* overrides this default. The VMS-style directory name *location* is used as the location of the Maple library instead. *location* also becomes the value of the Maple variable `libname`.

A.7.3. Suppress initialization option: -s

MAPLE `-s` will cause Maple to forego reading any initialization files (see the section "Maple initialization files" of this appendix).

A.7.4. Quiet option: -q

MAPLE `-q` will suppress the printing of Maple's startup logo and various informational messages ("words used" messages and garbage collection messages).

---

‡ These options, like I/O redirection, derive their form and syntax from the Unix and GCOS systems where Maple was originally developed.

A.8. MINT

Mint is a program which checks a Maple source file for syntax errors and gives warnings about possible problems. Additionally, it can report how variables are used within procedures defined in a Maple source file. The general functionality of the Mint program is explained in chapter 10. (See section 10.1.)

The most common syntax for the Mint command on VMS is:

`MINT` *input_file*

which invokes Mint to process the Maple procedures and commands contained in the Maple source file *input_file*. More generally, the amount of information reported by the Mint command is controlled by an *information level* flag as explained in Chapter 10. The syntax for specifying this flag is:

`MINT -I` *info_level input_file*

where *info_level* is an integer between 0 and 4.

There are additional optional arguments to the Mint command, with the general format:

`MINT` { `-I` *info_level* } { `-L` }
{ `-D` *library_database_file* } { `-Q` }
{ `-A` *database_file* } *input_file*

The braces indicate optional arguments. See HELP MINT on VMS for further details.

A.9. Summary of site- and VMS- dependent aspects of Maple

Site dependencies

Invoking Maple	Type MAPLE after seeing the VMS $ prompt. Check with your local system consultants if this doesn't work.
Owner of Maple library, related code	Typically, the MAPLE account.
Location of Maple library	Given by VMS logical-name MAPLE$LIB.
Location of Maple system code	Given by VMS logical-name MAPLE$SYSTEM.
Access to Maple code	Typically, users have read access to MAPLE$LIB files and execute access to MAPLE$SYSTEM files.

VMS dependencies

Interrupt (quit) signal (CTRL/Y)	Will exit Maple immediately. Use the VMS command CONTINUE to restart.
Cancel signal (CTRL/C)	Abort currently executing Maple command, but remain in Maple.
Escape to VMS	!*VMS_command* escapes Maple temporarily to perform the *VMS_command*. After the command finishes executing, control returns to the Maple session. See also the Maple built-in function **system**.
Maple file names	The corresponding VMS file name and type can be obtained via **convert**(..., *hostfile*).

Session initialization for all users	Defined in the [.SRC]INIT file of the Maple library. This file, if it exists, is typically established by the system manager.
User's own session initialization	Given by the value of VMS "Process logical" MAPLE$INI.
The Maple **words** function and "*words used*" messages	Words of memory are measured in units of 32 bits (4 bytes).
The Maple **time** function and "*time=* " messages	Time reflects CPU cycles devoted to Maple execution, measured in seconds.
Character set	The printable ASCII set (excludes control characters, etc.). *Note that upper and lower cases are distinguished within Maple* – `QUIT` is not the same as `quit`, etc.
Mint	`MINT` *input_file* `MINT -I` *info_level input_file* `MINT` { `-I` *info_level* } { `-L` } { `-D` *library_database* } { `-Q` } { `-A` *database_file* } *input_file*

B. Appendix B — Maple under the IBM VM/CMS operating system

B.1. Maple and CMS

Maple is not designed to be a completely self-contained programming environment; rather, it is designed to work with other services provided by the operating system of the computer on which it runs. Using Maple under the IBM VM/CMS operating system requires some knowledge of the fundamentals of CMS, such as logging on and off, using a text editor, and listing and deleting files. This section assumes that the reader knows how to do such activities on CMS without providing further explanation. The reader is referred to *The CMS User's Guide*[Sys79] for a basic overview of CMS usage.

The standard installation instructions for Maple make it possible for CMS users to start up Maple by typing MAPLE. The related program MINT should also be available as a system command. Information is available on-line via the CMS HELP command for Maple and Mint.

B.2. Maple initialization files

The file named INIT MAPLIB$ plays a special role for all Maple sessions under CMS. If that file exists (either on a "system disk" or on a "user disk") then it is read (in the sense of the Maple **read** statement) as the first action in any Maple session, before any Maple commands from the user are executed. The INIT MAPLIB$ file typically contains Maple commands establishing start-up values for Maple system variables, such as those controlled by **gc** and **words**. It is typically set up by the system manager to establish start-up values for such variables that are appropriate for the site.

If a user has a file named MAPLEINI FILE then when Maple is invoked, this file is read into Maple after the INIT MAPLIB$ file (see previous paragraph) is read in, supplementing or overriding the Maple commands found in the INIT MAPLIB$ file. However, many users find that they don't need a MAPLEINI FILE if the default start-up actions work satisfactorily for them.

B.3. Temporarily escaping from Maple

When interacting with Maple, typing a line beginning with ! means that what follows is to be interpreted and executed as a CMS subset command instead of a Maple statement or command. All CMS subset commands are accessible via this mechanism, e.g., !QUERY DISK or !TYPE MAPLEINI FILE . When execution of the CMS command is finished, the Maple session is resumed where it had been suspended. The results of the

command do not affect the state of the Maple session when it is resumed.

If the given command isn't part of the subset, then attempted execution of the command may abort the Maple session. The CMS subset varies among installations, but typically, simple CMS commands such as QUERY, LIST, or TYPE are in the subset.

B.4. Translating between Maple file names and CMS file names

Maple interacts with the CMS file system via the three Maple commands: **save**, **read**, and **readlib**. With both **save** and **read**, the user can specify the files to be written or read in the usual CMS-style; e.g.,

```
save x, `JHJ DATA A`;
```

However, it is possible with **save** and **read**, and obligatory with **readlib**, to use Maple-style file names; e.g., those used in chapters 1-10 of this *Reference Manual*. Maple-style names follow different rules than those of CMS.†.

In the Maple style of naming files, parts of the file name may be separated by /'s. File names, like all Maple names, may be created by concatenating several names through the Maple . (dot) operator.

There are two kinds of formats for files that Maple reads and writes: source (such as one can create by typing Maple commands as input to a text editor), and internal format (an encoded form that can be more quickly read into a Maple session). On CMS, as on other operating systems, Maple follows the rule that if a Maple-style file name ends in .m then the file is treated as an internal format file. Otherwise, it is treated as a source file†.

B.4.1. Translation rules

In the context of a **read**, **save**, or **readlib** command, "Maple-style" file names are automatically translated into CMS-style names as follows:

(1) If the Maple file name contains no slashes (/) or dots (.) before the first blank and it contains a blank, it is unchanged.

(2) The following template transformations are performed on the remaining part of the string up to the first blank or the end of the string:

---

† Maple file naming conventions are derived from the Honeywell GCOS and Unix operating systems, where Maple was originally developed. The style has been retained to allow Maple to be easily portable across several different kinds of operating systems.

LIB/src/<A>	→	<A>	MAPLIB$
LIB/<A>.m	→	<A>	MAPLIB$M
LIB/help/text/<A>	→	<A>	MAPLIB$H
LIB/<B>/src/<A>	→	<A>	<B>$
LIB/<B>/<A>.m	→	<A>	<B>$M
LIB/help/<B>/text/<A>	→	<A>	<B>$H
LIB/<C1>/<C2>/src/<A>	→	<A>	<C1><C2>$
LIB/<C1>/<C2>/<A>.m	→	<A>	<C1><C2>$M
<E>/<D1>.<D2>	→	<D1>	<D2> <E>
<A>.<D>	→	<A>	<D>
<D>	→	<D>	

where

<A>, <B>, <C1>, and <C2> match sequences of characters that are alphanumeric, $, @, #, or . (dot)

and

<D>, <D1>, and <D2> match sequences of characters that are alphanumeric, $, @, or #

and

<E> is a file mode (a single letter, or a single letter followed by a digit between 0 and 5).

For each of <A> through <E>, the following changes are made:

(i) Lower case letters are converted to upper case.

(ii) In the context of a LIB path name, if the first letter of the original Maple-style file name <A> was an upper case letter then the prefix $ is prepended to the resulting CMS file name <A>.

(iii) Extra dots are replaced by dollar signs ($).

(iv) <A>'s and <D>'s are truncated to 8 characters, counting the $ prefix which may have been added by rule (ii). Zero-length <A>'s and <D>'s are replaced by dollar signs ($). <B>'s are truncated to 6 characters. <C1>'s and <C2>'s are truncated to 3 characters.

If none of these templates matches the input then an error message is generated, of the general form, "*Error, unable to read ....*", or "*Error, unable to save....*".

(3) If the complete string was not analyzed in step (2), then the remainder of the string is added to the name. This allows file-modes and DCB options to be specified using the CMS conventions.

B.4.2. Examples

1. `read FilenameAlpha;` will read into Maple the CMS file FILENAME FILE on the user's default disk. The file will be read with the assumption that it is in source format.
2. `save a, `lgp.m`;` will save the value of the Maple variable `a` in Maple internal format, in the CMS file LGP M on the user's default disk.
3. `save a, `lgp.m a0`;` will save the value of the Maple variable `a` in Maple *source format* in the file LGP M A0.
4. `read ``.libname.`/euler.m`;` will read the file EULER MAPLIB$M from the user's disk. The intended effect is to read in the Maple library procedure **euler**; this will work only if there is no other file with that name encountered in the user's default search (i.e. on another disk). This example is useful only as an illustration of the file name conversion for library functions; most users would prefer to retrieve the **euler** function with the Maple command `readlib(euler)` .

B.4.3. Convert utility

Maple's **convert** library function can be used to discover how a Maple-style file name will be translated into a CMS-style name. Specifically, `convert(`*myfilename*`, hostfile)` will return the CMS name that Maple translates *myfilename* into.

Examples:

`convert( `esa.m`, hostfile)` will return as its value the name ESA M (BIN RECFM F LRECL 4.

`convert( `b/saq.m`, hostfile)` will return SAQ M B (BIN RECFM F LRECL 4.

B.5. Redirection of input and output

Invoking the MAPLE command with a CMS-style fileid as arguments results in the specified file being used as the input source of a Maple session instead of the terminal keyboard. Once all lines have been read from the file, additional input may be entered through the terminal keyboard (unless the session has ended because the file contained a **quit**, **done**, or **stop** command).

```
MAPLE  LGP FILE A
```

will use the contents of LGP FILE A as the input to the Maple session instead of the terminal keyboard.

Similarly, if Maple is invoked with two CMS-style fileids, then the first fileid will be used as the input file, and the second fileid will be the output file of the Maple session, which will not appear at the terminal. Output redirection must be used with input redirection.

```
MAPLE LPG FILE A LGPOUT FILE A
```

will read from the file LPG FILE A and place the results of the Maple session in the file LGPOUT FILE A.

```
MAPLE  BJKIN FILE A  LGPOUT FILE A
```

will start up a Maple session, with input coming from BJKIN FILE A, and results placed in the file LGPOUT FILE A. Note that LGPOUT FILE A will not contain the commands (from BJKIN FILE A), but only the results of commands.

B.6. MAPLE command line options for CMS

B.6.1. Overview of Maple command line options

Maple users typically invoke Maple by typing MAPLE. However, the Maple command has two options: SUPPRESS and QUIET. The options are used to change effects that occur by default during a Maple session, and are explained in the sections that follow.

Command line options can be used together with redirection of input and output from and to files, but then the command line options must appear after the (.

```
MAPLE  IN FILE A  OUT FILE A  ( QUIET
```

invokes Maple with the QUIET option, taking input from IN FILE A and placing results in OUT FILE A.

B.6.2. Suppress initialization option: SUPPRESS

`MAPLE ( SUPPRESS` will cause Maple to forego reading any initialization files (see the section "Maple initialization files" of this appendix).

B.6.3. Quiet option: QUIET

`MAPLE ( QUIET` will suppress the printing of Maple's startup logo and various informational messages ("words used" messages and garbage collection messages).

B.7. Prompt

Maple prints the prompt string > when it is expecting input. The user should wait for the prompt string to appear before entering the next input line. **On CMS, typing characters before the prompt appears will generate an interrupt signal for Maple and cause it either to stop its current computation or to drop out of the Maple session altogether.**

B.8. MINT

Mint is a program which checks a Maple source file for syntax errors and gives warnings about possible problems. Additionally, it can report how variables are used within procedures defined in a Maple source file. The general functionality of the Mint program is explained in chapter 10. (See section 10.1.)

The most common syntax for the Mint command on CMS is:

`MINT -I` *input_fileid*

which invokes Mint to process the Maple procedures and commands contained in the Maple source file *input_fileid*. More generally, the amount of information reported by the Mint command is controlled by an *information level* flag as explained in Chapter 10. The syntax for specifying this flag is:

`MINT -F` *info_level* `-I` *input_fileid*

where *info_level* is an integer between 0 and 4.

There are additional optional arguments to the Mint command, with the general format:

```
MINT { -F info_level } { -L }
                { -D library_database_fileid }
                { -A database_fileid } { -Q }
                -I input_fileid { -O output_fileid }
```

The braces indicate optional arguments. See HELP CMS MINT on CMS for an explanation of the optional arguments.

B.9. Summary of site- and CMS- dependent aspects of Maple

Site dependencies

Invoking Maple	Type MAPLE after seeing the CMS prompt. Check with your local system consultants if this doesn't work.
Maple variable *libname*	On CMS, the Maple variable *libname* simply contains the value LIB. The MAPLE EXEC file will link to a system disk which contains the Maple system, including the Maple library.
Access to Maple code	Typically, users have read access to the system disk files containing the Maple library, and execute access to those of the Maple system.

CMS dependencies

Escape to CMS	!*CMS_command* escapes Maple temporarily to perform the *CMS_command*, which must be in the "CMS subset of commands". After the command finishes executing, control returns to the Maple session. (See also the Maple built-in function **system**.) The "CMS subset" is site-dependent, but typically simple commands such as QUERY or LIST are part of it.
Maple file names	The corresponding CMS file name and type can be obtained via **convert**(..., *hostfile*).
Session initialization for all users	The file named INIT MAPLIB$, either on a system disk or a user disk, may contain initialization commands.

	This file, if it exists, is typically established by the system manager.
User's own session initialization	The file named MAPLEINI FILE on the user's disk, if it exists, overrides the system default initialization.
The Maple **words** function and "*words used*" messages	Words of memory are measured in units of 32 bits (4 bytes).
The Maple **time** function and "*time=* " messages	Time reflects CPU cycles devoted to Maple execution, measured in seconds.
Character set	The printable EBCDIC set. On EBCDIC terminals, the effect of the ^ character appearing in the Maple documentation can be achieved via the character ¬ (HEX 5F). Maple uses HEX AD for the left square bracket [, which may be entered using the two-character sequence (| , and HEX BD for the right square bracket] with corresponding two-character sequence |) . For the left brace { or HEX C0 use the two-character sequence (* , and for the right brace } or HEX D0 use the two-character sequence *) . *Note that upper and lower cases are distinguished within Maple* – QUIT is not the same as `quit`, etc.
Mint	MINT -I *input_fileid* MINT -F *info_level* -I *input_fileid* MINT { -F *info_level* } { -L } { -D *library_database_fileid* } { -A *database_fileid* } { -Q } -I *input_fileid* { -O *output_fileid* }

C. Appendix C – Maple under the UNIX operating system

C.1. Maple and Unix

Maple is not designed to provide a completely self-contained programming environment; rather, it is designed to work with other services provided by the operating system of the computer on which it runs. Using Maple under the Unix operating system requires some knowledge of the fundamentals of Unix usage, such as logging in and out, using a text editor, and listing and deleting files. This section assumes that the reader knows how to do such activities without providing further explanation. Such explanations can be found in any basic text on Unix, such as [Ker84].

The standard installation instructions for Maple make it possible for Unix users to start a Maple session by typing `maple`. The related program `mint` should also be available as a system command. Information is available on-line via the Unix `man` command, e.g. *man maple*.

C.2. Maple initialization files

The file *src/init* in the Maple library plays a special role for all Maple sessions under Unix. If that file exists, it is read (in the sense of the Maple **read** statement) as the first action in any Maple session, before any Maple commands from the user are executed. The *init* file typically contains Maple commands establishing start-up values for Maple system variables, such as those controlled by **gc** and **words**. It is typically set up by the system manager to establish start-up values for such variables that are appropriate for the site.

If a user has a file named *.mapleinit* in his or her home directory, that file is read into Maple after the system-wide *src/init* file. The user can put Maple commands into *.mapleinit* that supplement or override those found in the *src/init* file. However, many users will find that they don't need a *.mapleinit* file if the default start-up actions work satisfactorily for them.

C.3. Quit and Interrupt characters

Maple uses the user's Unix quit (for many users, *control-*) and interrupt (for many users, *delete* or *break*) signals to interrupt a Maple calculation. If the quit character is typed, the Maple session will be aborted, and control returns to the user's shell. If the interrupt character is typed once, the currently executing Maple calculation will be aborted, and another Maple command can then be entered. If the interrupt character is typed twice within three seconds, the Maple session will be aborted as though an

interrupt character had been typed. On Berkeley Unix systems, the "suspend" character (typically, *control-Z*) works with Maple as with other foreground processes.

C.4. Temporarily escaping from Maple

When interacting with Maple, typing a line beginning with `!` means that what follows is to be interpreted and executed as a Unix command instead of a Maple statement or command. All Unix commands are accessible via this mechanism, e.g., `!vi` or `!who`. When execution of the Unix command is finished, the Maple session is resumed where it had been suspended. The results of the command do not affect the state of the Maple session when it is resumed.

C.5. Redirection of input and output

Maple on Unix uses the standard input and output files for reading and printing information. By default, these direct input from and output to the user's terminal. One can change this by employing the usual means of the Unix command language. Invoking the `maple` command with a `<` and a file name results in the specified file being used as the input source of a Maple session instead of the terminal keyboard. Once all lines have been read from the file, additional input may be entered through the terminal keyboard (unless the session has ended because the file contained a Maple **quit**, **done**, or **stop** command).

Similarly, if Maple is invoked with `>` and a file name, then the results of the Maple session will be placed in the specified file instead of the terminal. Input redirection and output redirection may be used together.

Example:

```
maple < file
```

will use the contents of `file` as the input to the Maple session instead of the terminal keyboard.

```
maple < input_file > output_file
```

will start up a Maple session, take input from `input_file`, and place the results of the Maple session in `output_file`. Note that `output_file` will not contain the commands (from `input_file`), but only the results of commands.

C.6. `maple` command line options for Unix

C.6.1. Overview of Maple command line options

Maple can be invoked with up to three options on the command line: *-s*, *-q*, and *-b*. The effect and purpose of each of them is described in the following subsections. Users will not need to include any of them on the command line for typical interactive usage of Maple.

Command line options can be used together with redirection of input and output from and to files, but then the command line options must appear before the < or >.

Example:

```
maple -q < input_file > output_file
```

invokes Maple with the *-q* option, taking input from `input_file` and placing results in `output_file`.

C.6.2. Library specification option: -b

The Maple variable *libname* names the default location of the Maple library directory. The command `maple -b` *location* invokes Maple where this default is overridden, using *location* as the location (directory) of the Maple library instead. *location* replaces the default value of the Maple variable `libname` for that session.

C.6.3. Suppress initialization option: -s

`maple -s` will cause Maple to forego reading any initialization files (see the section "Maple initialization files" of this appendix).

C.6.4. Quiet option: -q

`maple -q` will suppress the printing of Maple's startup logo and various informational messages ("words used" messages and garbage collection messages).

C.7. `mint`

Mint is a program which checks a Maple source file for syntax errors and gives warnings about possible problems. Additionally, it can report how variables are used within procedures defined in a Maple source file. The general functionality of the Mint program is explained in chapter 10. (See section 10.1.)

The most common syntax for the Mint command on Unix is:

`mint` *input_file*

which invokes Mint to process the Maple procedures and commands contained in the Maple source file *input_file*. More generally, the amount of information reported by the Mint command is controlled by an *information level* flag as explained in Chapter 10. The syntax for specifying this flag is:

`mint -i` *info_level input_file*

where *info_level* is an integer between 0 and 4.

There are additional optional arguments to the Mint command, with the general format:

`mint` { `-i` *info_level* } { `-l` } { `-d` *library_database* }
{ `-a` *database_file* } { `-q` } { *input_file* }

The braces indicate optional arguments. See *man mint* on Unix for an explanation of the optional arguments.

C.8. Summary of site- and Unix- dependent aspects of Maple

Site dependencies

Invoking Maple	Type `maple` after seeing the Unix prompt. Check with your local system consultants if this doesn't work.
Owner of Maple library, related code	Ask your system manager for information about your site, or use the *ls -l* command on the Maple library directory.
Location of Maple library	Given by Maple variable `libname`.
Location of Maple system code	Ask your system manager for information about your site.
Access to Maple code	Typically, users have read access to Maple library files and execute access to `maple` and `mint`.

Unix dependencies

Quit signal (control-\ on some Unix systems)	Will exit Maple immediately.
Interrupt signal (*delete* or *break* on some Unix systems)	Abort currently executing Maple command, but remain in Maple.
Escape to Unix	`!`*Unix_command* escapes Maple temporarily to perform the *Unix_command*. After the command finishes executing, control returns to the Maple session. (See also the Maple built-in function **system**.)
Maple file names	Maple file names are the same style as Unix file names. `convert(` *filename*`, hostfile)` on Unix sys-

	tems returns its *filename* argument without transformation.
Session initialization for all users	Defined by the system manager in the file *src/init* of the Maple library.
User's own session initialization	*.mapleinit* in the user's home directory.
The Maple **words** function and "*words used*" messages	Words of memory are measured in units of 32 bits (4 bytes).
The Maple **time** function and "*time=* " messages	Time reflects CPU cycles devoted to Maple execution, measured in seconds.
Character set	All ASCII characters (including control characters, etc.)
Mint	**mint** *input_file* **mint** `-i` *info_level input_file* **mint** { `-i` *info_level* } { `-l` } { `-d` *library_database* } { `-q` } { `-a` *database_file* } { *input_file* }

References

Bou71.
S.R. Bourne and J.R. Horton, The Design of the Cambridge Algebra System, pp. 134-143 in *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ed. S.R. Petrick, Special Interest Group on Symbolic and Algebraic Manipulation, Association for Computing Machinery (1971).

Cha88.
B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt, *First Leaves: A Tutorial Introduction to Maple, 2nd edition*, WATCOM Publications Ltd., Waterloo, Ontario, Canada (1988).

Cor82.
Digital Equipment Corporation, *VAX/VMS Primer*, Order No. AA-D030C-TE. May 1982.

Hal71.
Andrew D. Hall, Jr., The ALTRAN System for Rational Function Manipulation - A Survey, pp. 153-157 in *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ed. S.R. Petrick, Special Interest Group on Symbolic and Algebraic Manipulation, Association for Computing Machinery (1971).

Hea71.
Anthony C. Hearn, Reduce 2: A System and Language for Algebraic Manipulation, *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, pp. 128-133 Association for Computing Machinery, (1971).

Ker84.
Brian W. Kernighan and Rob Pike, *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs, New Jersey (1984).

Mar71.
W.A. Martin and R.J. Fateman, The MACSYMA System, pp. 59-75 in *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ed. S.R. Petrick, Special Interest Group on Symbolic and Algebraic Manipulation, Association for Computing Machinery (1971).

Sys79.
IBM Systems, *IBM Virtual Machine Facility/370: CMS User's Guide, 3rd edition*, Order No. GC20-1819-2. 1979.

Index

! 23, 368, 370, 378, 387
$ 20
& 25
& name 25
& operator 25
&^ 24, 265
-b option 373, 388
-q option 373, 388
-s option 373, 388
< 26, 373, 388
<= 26
<> 26
= 26
> 26, 373, 388
>= 26
@ 24
@@ 24, 34
aborting Maple, under Unix 386
aborting Maple, under VMS 370
abs 70, 116
actual parameters 76
add 281, 317
addcol 282
addition 23, 45
addrecord 339
addrow 282
adjoint 282
algebraic 46
algebraic context 26
algebraic operator 23
alloc 363
alternate symbols 3
anames 116
and 27, 34, 47
angle 283
angle brackets 32
antisymmetric 68
application 94
arc-hyperbolic functions 116
arc-trigonometric functions 116
arccos 116
arccosh 116
arccot 116
arccoth 116
arccsc 116
arccsch 116
arcsec 116
arcsech 116
arcsin 116
arcsinh 116
arctan 116
arctanh 116
args 76, 86
array 29, 49, 56, 57, 58, 111, 116, 125
assign 117
assigned 113, 118
assignment 10, 59
asympt 118
average 340
back quote 7
backslash 7, 14, 18
band 283
base 126
basis 327
bernoulli 119
bezout 283
binary 126
binomial 119
blank 6
boolean 47, 150
boolean context 27, 47
boolean expression 47
boolean procedure 87
break 12
break/interrupt 13
Brillhart 162
bspline 230
builtin 79
builtin functions 113
C 230
calculus package 348

carriage return 6
case statement 11
cat 120
Catalan 151
centralizer 271
cfrac 306
changevar 351
character set 3, 376, 385, 391
chebyshev 120
ChiSquare 336
chrem 230
Ci 231
circular functions 216
close 259
cmpltsq 352
CMS 377
CMS subset of commands 378
coeff 121
coeffs 121
coeftayl 231
coldim 284
collect 122
colon 6
colspace or range 284
combine 352
command line options 367
comment 6
commutators 232, 303
complex numbers 150
compoly 233
compose 318
composition 94
composition operator 24
concatenation 18, 28, 52, 120, 370, 378
cond 284
conditional statement 11
confrac 127
conj 303
constant 14
constant operators 33
constants 365
Content 123
content 122
continuation 6, 14, 18
continued fraction 127
convergs 233
convert 8, 89, 114, 124, 371, 380
convert/disjcyc 272
convexhull 328
copy 65, 141
correlation 340
cos 216
cosets 272
cosh 161
cosrep 273
cot 216
coth 161
covariance 340
crossprod 285
csc 216
csch 161
cterm 328
curl 285
cyclotomic 306
D 96, 234, 302, 348
d 261
DAG 110
data types 42, 101
debugging 358, 361
decimal 128
defform 261
define 141
definite 286
degree 144
degrees 128
denom 178
denominator 178
det 286
diagonal 68
diff 110, 144, 353
differential equations 146
differential forms 260
differentiation 91, 144

differentiation operator 96
difforms package 260
Digits 14, 43, 151, 365
dilog 234
dimension 56
dinterp 235
directed acyclic graph 110
directional limits 170
discrim 235
distance 353
ditto 23
diverge 287
Divide 145
divide 110, 145
division 23
divisors 306
do statement 12
done 13
dotprod 287
dsolve 146
dual 328
E 151
easy 162
edit 235
Ei 236
eigenvals 287
ellipsis 27
empty list 21
empty set 21
empty statement 13
eqn 237
eqnlist 128
eqns 303
equality 129, 329
equation 26, 46, 47
erf 147
ERROR 86, 147
error, syntax 358
escape to host 368
euler 148
eulermac 237
Eval 349
eval 113, 148
evala 149
evalb 47, 150
evalc 150
evalf 14, 43, 110, 151
evalm 238
evaln 113, 152
evalpow 318
evalstat 341
evaluation 30, 60, 74, 78
evaluation of arguments 113
evaluation of locals 78
evaluation, one-level 78
exit from Maple 364
exp 129, 152
expand 110, 153
Exponential 337
exponentiation 23, 34, 45
expression 13
expression sequence 20, 28, 29, 48
expression, latest 23
exprseq 48
expsincos 130
F 305
Factor 154
factor 153
factorial 23, 51, 130
factorset 307
fail return 86
false 27, 47
Fdist 337
feasible 329
fibonacci 238
files 8
finitely-presented groups 271
fixdiv 239
Float 15, 43
float 43, 130, 151
floating-point number 14, 43
for statement 12
forget 81, 239
formal parameters 75

formal power series 317
formpart 263
fortran 240
frac 216
fraction 14, 42, 137
freeze 240
frontend 110, 154
fsolve 155
Ftest 337
function 30, 31, 51
function assignment 80
functional operators 32
G 313
GAMMA 131, 156
gamma 151
garbage collection 364, 369, 377, 386
gausselim 288
gbasis 267
gc 156, 364, 365, 369, 377, 386
Gcd 157
gcd 157
Gcdex 159
gcdex 158
genmatrix 288
genpoly 159
getkey 341
global variables 365
grad 289
grammar 52
graphics 186
grelgroup 274
Grobner basis package 267
group 142
group package 271
groupmember 274
grouporder 275
H 151, 314
hadamard 289
has 160
hash table 80
hashing 107
hastype 160
help 2, 161
hessian 289
hex 131
hilbert 290
horner 131
Horner form 131
hostfile 8, 89, 114, 132, 371, 380
hyperbolic functions 161
hypergeom 241
hypergeometric function 241
I 150
IBM 377
icontent 161
identities 204
identity 68
identity operator 33
if statement 11
ifactor 162
igcd 163
igcdex 163
ilcm 163
imagunit 307
indets 163
index bound 56
indexed 19, 44
indexed name 44, 61
indexfunc 290
indexing function 67
indices 164
inequalities 205
inequality 26, 46, 47
infinity 171
infinity plots 182
INIT file (CMS) 377
INIT file (VMS) 369
initialization files 367, 369, 377, 386
Int 349
int 28, 164
integer 14, 42
integer factorization 162

integration 164
inter 275
intercept 354
internal format 8, 9, 88, 370, 378
internal functions 99
internal organization 99
Interp 166
interp 165
interrupt character, under Unix 386
interrupting Maple computations, under Unix 386
interrupting Maple computations, under VMS 370
intersect 22
intparts 354
inverse 290, 319
inverse Laplace transform 241
invlaplace 241
invperm 275
iquo 166
irem 166
iroot 242
iscont 242
ismith 291
isnormal 276
isolate 243, 354
isolve 167
isprime 167
isqrfree 307
isqrt 168
issqr 243
ithprime 168
jacobi 307
jacobian 291
kernel or nullspace 292
keyword 4
L 314
lambda 308
lambda calculus 94
laplace 244
Laplace transform 244
laplacian 292
last name evaluated 60, 74
lasterror 86, 215, 366
latex 244
lcm 157
lcoeff 168
ldegree 144
leadmon 268
leastsqrs 292
legendre 308
length 169
lessequal 129
lessthan 129
lexorder 169
libname 89, 114, 366, 367, 373, 388
library 9, 68, 89
library functions 113
library specification option 367, 373, 388
Limit 350
limit 170
linalg package 281
line 6
line feed 6
line plots 182
linear 143
linear algebra 281
linear equations 205
linear optimization 327
linregress 341
linsolve 293
list 21, 29, 49, 132
listlist 133
ln 133, 172
load 9
load, automatic 9, 67, 90
local variables 78
log 365
logical operator 26, 27, 34
lower case, characters 376, 385, 391

lprint 172
M 305
makeproc 355
man command, under Unix 386
map 75, 173
MAPLE$INI (VMS) 369
MAPLE$LIB (VMS) 369
MAPLEINI FILE (CMS) 377
mapleinit file (Unix) 386, 391
Margay 112
match 245
max 173
maximize 330
maxnorm 174
mcombine 308
mdet 293
median 342
member 174
memory 363, 376, 385, 391
metric 134
midpoint 356
min 173
minimize 330
minimize and maximize 246
Mint 358, 374, 382, 389
minus 22
minus, unary 23
mixpar 263
mlog 309
mobius 309
mod 24, 174, 365
mode 342
modp 174
mods 174
Morrison-Brillhart 162
mroot 309
msolve 176
msqrt 310
mtaylor 246
mulcol 294
mulperms 276
mulrow 294
multconst 319
multidirectional limits 171
multiple plots 182
multiplication 23, 45
multiply 294, 319
multregress 343
N 338
name 5, 16, 29, 30, 44, 45, 134
name table 109
nargs 77
natural integer 5
negative 320
neutral operator 25
newline 6
Newman-Penrose formalism 302
next 12
nextprime 177
non-commutative multiplication 91
nonneg 331
nops 42, 178
norm 295
normal 110, 177
normalf 269
normalizer 276
not 27, 47
np package 302
NULL 20, 48, 56, 61, 62
null string 89
number theory 305
numer 178
numerator 178
numeric 14, 44
numtheory package 305
O 46, 179, 213
octal 134
one-level evaluation 44, 78
op 28, 42, 43, 44, 48, 49, 51, 56, 178
open 259
operator 4, 79
operator definition 91
operator precedence 34

optimize 247
option operator 32
option remember 80
option system 81
options 79
or 27, 34, 47
Order 213, 365
order 179
order term 46, 179, 213
orthog 295
orthogonal polynomials 313
orthopoly package 313
P 315
packages 113, 114
Pade approximant 137
parameter passing 76
parametric plots 184
parfrac 135
parity 263
parser 99
partial fraction expansion 135
permanent 296
permgroup 277
permrep 278
permutation groups 271
permute 248
Pfaffian operators 302
phi 310
Pi 151
pivot 331
pivoteqn 332
pivotvar 332
plot 179
plot device 180
plot options 183
plot ranges 185
plus, unary 23
point plots 184
polar 135
polar coordinates 184
Pollard 162
polynom 136
polynomial equations 202
portability 111
powcreate 320
powdiff 321
powexp 321
powint 321
powlog 322
powpoly 322
powseries 317
powsolve 323
powsubs 356
pprimroot 310
precedence of operators 34
Prem 189
prem 188
pres 278
prettyprint 1, 365
prevprime 177
Primpart 123
primpart 122
primroot 311
print 190
printlevel 2, 361, 365
priqueue 248
procedure 30, 50, 74
procname 77, 86
product 190
profile 249, 365
programming-language operator 4
projection 343
prompt 35, 365
proot 250
properties 143
Psi 191
psqrt 250
punctuation mark 6
put tables 110
putkey 344
QUIET option 382
quiet option 367, 373, 388
quit 7, 13, 364
quit character, under Unix 386

quit character, under VMS 370
Quo 193
quo 193
quote (double) 23
quote (single) 30, 52
quotient 323
radians 136
radical 136, 206
radsimp 191
rand 191
random numbers 191
randpoly 250
range 27, 47
rank 296
ratio 333
rational 43, 137
rational function 137
ratpoly 137
read 9, 10, 88, 369, 370, 377, 378, 386
readlib 67, 89, 110, 192, 370, 378
readlib-defined functions 113
readstat 193
realroot 251
redirection, of input and output 371, 381, 387
regression 344
relation 26, 46, 47
Rem 193
rem 193
remember 79
remember table 80, 110
removekey 345
repeated composition 34
repeated composition operator 24
repetition statement 12
replot 252
reserved word 4
residue 252
Resultant 195
resultant 194
RETURN 86, 195
reversion 324
rho method 162
RootOf 138, 196
rootsunity 311
round 216
rowdim 296
rowspace 296
rsolve 196
Rsquared 338
S 253
safeprime 312
save 9, 10, 89, 370, 378
scalarmul 297
scalarpart 264
screenheight 365
screenwidth 365
sdev 345
search 253
sec 216
sech 161
select 253
selection operation 29
selection statement 11
semicolon 6
sequence operator 20
series 45, 139, 179, 207, 213
serr 345
session 35
set 21, 29, 49, 139
set operator 22
setup 333
Shanks 162
showprofile 249
showtime 254, 365
Si 254
sign 197
signature 109
signum 198
simpform 265
simplex package 327
simplification 30, 108
simplification of operators 33

simplification table 108
simplify 198
simplify/atsign 199
simplify/GAMMA 199
simplify/hypergeom 200
simplify/power 200
simplify/Psi 200
simplify/radical 201
simplify/RootOf 201
simplify/sqrt 201
simplify/trig 202
sin 216
sincos 140
singular 255
singularvals 297
sinh 161
sinterp 255
slope 357
smith 297
solve 202
sort 208
source codes 114
source format 370, 378
space 6, 363
sparse 68
sparsymm 68
special characters 3
spline plots 187
spoly 270
Sprem 189
sprem 188
sqrfree 140
sqrt 209
square-free form 140
squfof 162
src/init, under Unix 386
standardize 334
statement 10
statement evaluator 99
statement separator 2, 6, 7
statistics 336
statplot 346
stats package 336
status 364, 366
std 334
stdle 334
stir1 256
stir2 256
stop 13
string 5, 16, 17, 44
student package 348
StudentsT 339
suball 304
subgrel 279
submatrix 298
subs 209
subsop 210
substring 210
subtract 325
subtraction 23
subvector 298
Sum 351
sum 28, 211
suppress initialization option 367, 373, 382, 388
SUPPRESS option 382
suspend character, under Berkeley Unix 387
swapcol 299
swaprow 299
sylvester 299
symmetric 68
symmpoly 256
syntax 52
syntax errors 358
system 79, 211
systems of equations 207
T 316
tab 6
table 19, 29, 49, 56, 57, 58, 111, 212
tan 216
tanh 161
tau 312

MAPLE

Reference Manual

5th Edition

Bruce W. Char
Keith O. Geddes
Gaston H. Gonnet
Michael B. Monagan
Stephen M. Watt

Brooks/Cole Publishing Company

0-921329-15-6
Printed in Canada
05/88